Software Intensive Systems Data Quality and Estimation Research in Support of Future Defense Cost Analysis

Final Scientific Technical Report SERC 2012-TR-032-1

March 13, 2012

Dr. Barry Boehm - University of Southern California

Contract Number: H98230-08-D-0171

Report No. SERC-2012-TR-032
March 16, 2012
UNCLASSIFIED

DO 02, TTO 02; RT 6

This material is based upon work supported, in whole or in part, by the U.S. Department of Defense through the Systems Engineering Research Center (SERC) under Contract H98230-08-D-0171. SERC is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology

SUMMARY

The accompanying report, "AFCAA Software Cost Estimation Metrics Manual," constitutes the 2011-2012 Annual Technical Report and the Final Technical Report of the SERC Research Task RT-6: Software Intensive Systems Data Quality and Estimation Research In Support of Future Defense Cost Analysis.

The overall objectives of RT-6 were to use data submitted to DoD in the Software Resources Data Report (SRDR) forms to provide guidance for DoD projects in estimating software costs for future DoD projects. In analyzing the data, the project found variances in productivity data that made such SRDR-based estimates highly variable. The project then performed additional analyses that provided better bases of estimate, but also identified ambiguities in the SRDR data definitions that enabled the project to help the DoD DCARC organization to develop better SRDR data definitions.

The resulting Manual provides the following guidance elements for software cost estimation performers and users. These have been reviewed and iterated over several performer and user workshops. Chapter 1 provides an overview, including discussion of the wiki form of the Manual. Chapter 2 provides consensus definitions of key software cost-related metrics, such as size, effort, and schedule. Chapter 3 compares the leading cost estimation models used in DoD software cost estimates -- COCOMO II, SEER-SEM, True S, and SLIM – in terms of the comparative inputs and outputs. Chapter 4 summarizes the content of the SRDR reports on DoD software projects. Chapter 5 provides definitions of the various DoD software applications domains used to help develop more representative estimates. Chapter 6 includes guidance for estimation in several frequently-occurring situations. These include relations among different metrics; effects of software reuse; estimation for COTS-based systems, particularly involving Enterprise Resource Planning (ERP) packages; and discussions of estimation for increasingly frequent emerging trends such as evolutionary development, net-centric systems of systems, model-based software engineering, and agile methods. Several appendices provide further related information on acronyms, sizing, nomograms, work breakdown structures, and references.

This Page Intentionally Left Blank

From AFCAA Software Cost Estimation Metrics Manual

AFCAA Software Cost Estimation Metrics Manual

Contents

Overview

The purpose of this manual is to help analysts and decision makers develop accurate, easy and quick software cost estimates for avionics, space, ground, and shipboard platforms. It was developed by the Air Force Cost Analysis Agency (AFCAA) in conjunction with service cost agencies, and assisted by the University of Southern California and the Naval Postgraduate School to improve quality and consistency of estimating methods across cost agencies and program offices through guidance, standardization, and knowledge sharing.

Metrics Definitions for consistent and repeatable measurements are pre-requisite to meet these goals. This section establishes a solid basis for cost estimation measures by providing standard and precise definitions. These are critical to handle different classes of software across diverse DoD environments for consistent measurement, data analysis and estimation. Checklists are provided so the measures can be interpreted unambiguously in different scenarios, and to document necessary adaptations or differences.

Cost estimation models widely used on DoD projects are overviewed next. This section describes the parametric software cost estimation model formulas, size inputs, cost drivers and how they relate to the standard Metrics Definitions. The models include COCOMO, SEER-SEM, SLIM, and True S. The similarities and differences for the cost model inputs (size, cost factors) and outputs (phases, activities) are identified for comparison.

The Software Resource Data Report (SRDR) is the form used for data collection and analysis on DoD projects. The required information for the Metrics Definitions is described in detail with examples. Procedures for access and usage of the DCARC database are provided. This database is the one used for DoD-wide analysis of the project cost metrics.

A quantitative Domain Analysis of the existing SRDR database is presented. The derived cost estimation relationships (CERs) and schedule estimating relationships (SERs) consistent with the Metrics Definitions are segmented by DoD domains and environments. These CERs and SERs can be used in developing estimates along with guidance from related metrics analysis on phase and activity effort percentages by domain.

Metrics Guidance culminates the manual. It includes data normalization guidelines conforming to the Metrics Definitions, how to use the Domain Analysis results, default productivity metrics and cost model inputs, guidelines for estimating growth and assessing schedule, estimation challenges for modern systems, and overall lessons learned.

It examines rules for estimating software size based on available data (requirements, use cases, etc.). It incorporates historical and subjective factors for quantifying effective size based on domain and adaptation type. Guidance for different classes of software covers COTS, open source and ERP projects. Software development and maintenance estimation are both addressed. Detailed examples are provided throughout such as computing equivalent size in different scenarios and handling multiple increments.

Appendices include Acronyms, Cost Model Factors, nomograms for visual computations of cost models and domain analysis estimating relationships, MIL-STD-881CWBS Mapping to Productivity Types, details of the statistical analyses, and references for all citations.

PDF Downloads

The complete manual in PDF format can be downloaded here. Any page can also be downloaded with the "Print/export" navigation tool.

Notes For Reviewers

Please add your ideas, suggestions or improvements on the Discussion pages with each chapter.

- You will need to **create a reviewer account** first.
- You will also be interested in reading **reviewer tips** for inserting your ideas, suggestions or improvements.

This wiki is still evolving. The Authors are finishing the material and will update the content based on Review feedback. The Author To-Do Lists are shown on the Discussion page.

Metrics Definitions

Size Measures

This section defines software product size measures used as input to cost estimation models and productivity analysis. Size measures are defined for consistency in the manual and interpreted for the selected models. Transformations between the respective model size measures are provided so projects can be represented in all models in a consistent manner and to help understand why estimates may vary. Guidelines for estimating software size and setting model size parameters using these definitions are provided in Metrics Guidance.

An accurate size estimate is the most important input to parametric cost models. However, determining size can be challenging. Projects may be composed of new code, code adapted from other sources with or without modifications, automatically generated or translated code, commercial software, and other types.

For estimation and productivity analysis, it is also desirable to have consistent size definitions and measurements across different models and programming languages. Consistent definitions must be provided for the different types of software to be meaningful distinctions and useful for estimation and project management. The basic definitions are provided next, elaborated for the Cost Estimation Models, and then applied in Domain Analysis and Metrics Guidance.

Source Lines of Code (SLOC)

The common measure of software size used in this manual and the cost models is Source Lines of Code (SLOC). SLOC are logical source statements consisting of data declarations and executables. Other size measures are sometimes collected and are defined later in this section.

SLOC Type Definitions

The core software size type definitions used throughout this manual are summarized in the table below. These apply to size estimation, data collection, productivity analysis, and guidance for setting cost model parameters. Some of the size terms have slightly variant interpretations in the different cost models as described in Cost Estimation Models.

Software Size Types

Size Type	Description
New	Original software created for the first time.
Adapted	Pre-existing software that is used as-is (Reused) or changed (Modified).
Reused	Pre-existing software that is not changed with the adaption parameter settings: ■ Design Modification % (DM) = 0% ■ Code Modification % (CM) = 0%
Modified	Pre-existing software that is modified for use by making design, code and/or test changes: ■ Code Modification % (CM) > 0%
Equivalent	A relative measure of the work done to produce software compared to the code-counted size of the delivered software. It adjusts the size of software relative to developing it all new. This is also sometimes called Effective size.
Generated	Software created with automated source code generators. The code to include for equivalent size may consist of the generator statements directly produced by the programmer, or the 3GL generated statements produced by the automated tools.
Converted	Software that is converted between languages using automated translators.
Commercial Off-The-Shelf Software (COTS)	Pre-built commercially available software components. The source code is not available to application developers. It is not included for equivalent size. Other unmodified software not included in equivalent size are Government Furnished Software (GFS), libraries, operating systems and utilities.

The size types are applied at the component level for the appropriate system-of-interest. If a component, or module, has just a few lines of code changed then the entire component is classified as Modified even though most of the lines remain unchanged. The total product size for the component will include all lines.

Open source software may be handled as other categories of software, depending on the context of its usage. If it not touched at all by the development team it can be treated as a form of COTS or reused code. However, when open source is modified it can be quantified with the adaptation parameters for modified code and be added to the equivalent size. The case for when you have source is shown in the last row of Table 17. Because you have more insight into internals, you test more thoroughly. The costs of integrating with open source with other software components should be added into overall project costs.

SLOC Counting Rules

Logical Lines

The common measure of software size used in this manual and the cost models is Source Lines of Code (SLOC). SLOC are logical source statements consisting of data declarations and executables. Table 5 shows the SLOC definition inclusion rules for what to count. Based on the Software Engineering Institute (SEI) checklist method [Park 1992, Goethert et al. 1992], each checkmark in the “Includes” column identifies a particular statement type or attribute included in the definition, and vice-versa for the “Excludes”.

Equivalent SLOC Rules for Development

	Includes	Excludes
Statement Type		
Executable	✓	
Nonexecutable		
Declarations	✓	
Compiler directives	✓	
Comments and blank lines		✓
How Produced		
Programmed New	✓	
Reused	✓	
Modified	✓	
Generated		
Generator statements		✓
3GL generated statements	✓ (development)	✓ (maintenance)
Converted	✓	
Origin		
New	✓	
Adapted		
A previous version, build, or release	✓	
Unmodified COTS, GFS, library, operating system or utility		✔ ✓

All of the models described calculate effort using the same size measure of logical source statements consisting of executables and data declarations. Other measures of size and their definitions follow.

Physical Lines

Physical source statements are source lines of code: one physical line equals one statement. The delimiter, or terminator, for physical source statements is usually a special character or character pair such as newline or carriage return. When dealing with source code files, the particular delimiter used depends on the operating system. It is not a characteristic of the language.

Total Lines

Total lines include everything including blank lines. Non-Commented Source Statements Only source statement lines, no blank lines, no comment-only lines.

To prevent confusion in reporting measures of size and in storing results in databases, counts of physical and logical source statements should always be maintained and reported separately.

Equivalent Size

The key element in software size for effort estimation is the equivalent (effective) size of the software product. Determining size can be approached from several directions depending upon the software size measure (lines of code, function points, use cases, etc.) used by

the development organization. A system developed by writing lines of code requires a different estimating approach than a previously developed or off-the-shelf application. The acquisition phase also influences the analyst's approach because of the amount and type of software development data available from the program or developers. [NCCA-AFCAA 2008]. Equivalent size is a quantification of the effort required to produce a software product. It may not be correlated to the total product size.

The guidelines in this section will help the estimator in determining the total equivalent size. The models/tools used for the estimate may not allow all the underlying inputs, such as for non-traditional size categories (e.g. a model may not provide inputs for auto-generated code). In these cases the estimator can calculate the equivalent size outside of the tool and provide that total as New size.

Definition and Purpose in Estimating

In addition to newly developed software, adapted software that is modified and reused from another source and used in the product under development also contributes to the product's equivalent size for cost models. A method is used to make new and adapted code equivalent so they can be rolled up into an aggregate size estimate. All of the models use equivalent size for calculating effort.

Equivalent size is also called "effective size" in the SEER and SLIM models, and the terms are interchangeable in the context of this manual. Both equivalent and effective refer to an adjustment of total size to reflect the actual degree of work involved.

The size of reused and modified code is adjusted to be its equivalent in new code for use in estimation models. The adjusted code size is called Equivalent Source Lines of Code (ESLOC). The adjustment is based on the additional effort it takes to modify the code for inclusion in the product taking into account the amount of design, code and testing that was changed and is described in the next section.

There are also different ways to produce software that complicate ESLOC including generated and converted software. All of the categories need to aggregated for equivalent size. A primary source for the equivalent sizing principles in this section is Chapter 9 of [Stutzke 2005].

A user perspective on Equivalent SLOC (ESLOC) described in [Stutzke 2005] provides a framework on what to include. Equivalent is a way of accounting for relative work done to generate software relative to the code-counted size of the delivered software. The source lines of code are the number of logical statements prepared by the developer and used to generate the executing code.

For usual Third Generation Language (3GL) software such as C or Java, count the logical 3GL statements. For Model-Driven Development (MDD), Very High Level Languages (VHLL), or macro-based development count the statements that generate customary 3GL code. In maintenance however, all the statements have to considered. Above the 3GL level, count the generator statements. For maintenance at the 3GL level, count the generated 3GL statements Two other primary effects that increase equivalent size are volatility and adaptation. Volatility is the percent of ESLOC reworked or deleted due to requirements volatility. Adaptation includes modified or reused as previously defined. These inclusion rules for ESLOC from the user perspective are embedded in the definitions for this manual. A summary of what to include in Equivalent Size for development is in the table below.

Equivalent SLOC Rules for Development

Source	Includes	Excludes
New	✓	
Reused	✓	
Modified	✓	
Generated		
Generator statements	✓	
3GL generated statements		✓
Converted		
COTS		✓
Volatility	✓	

Adapted SLOC Adjustment Factors

The normal Adaptation Adjustment Factor (AAF) is computed as:

$$AAF = .4\ DM + .3\ CM + .3\ IM$$

Where

% Design Modified (DM)
The percentage of the adapted software's design which is modified in order to adapt it to the new objectives and environment. This can be a measure of design elements changed such as UML descriptions.

% Code Modified (CM)
The percentage of the adapted software's code which is modified in order to adapt it to the new objectives and environment.

% Integration Required (IM)
The percentage of effort required to integrate the adapted software into an overall product and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

The AAF factor is applied to the size of the adapted software to get its equivalent size. The cost models have different weighting percentages as identified in Cost Estimation Models.

Reused software has DM = CM = 0. IM is not applied to the total size of the reused software, but to the size of the other software directly interacting with it. It is frequently estimated using a percentage. *Modified* software has CM > 0.

AAF assumes a linear effort relationship, but there are also nonlinear effects. Data indicates that the AAF factor tends to underestimate modification effort [Selby 1988], [Boehm et al. 2001], [Stutzke 2005]. Two other factors used to account for these effects are Software Understanding and Unfamiliarity:

Software Understanding
Software Understanding (SU) measures how understandable is the software to be modified. The SU increment is expressed quantitatively as a percentage. SU is determined by taking an average of its ratings on structure, applications clarity, and self-descriptiveness according to the Cost Factor Appendix.

Programmer Unfamiliarity
Unfamiliarity (UNFM) quantifies how unfamiliar with the software to be modified is the person modifying it. The UNFM factor described is applied multiplicatively to SU to account for the familiarity. For example, a person who developed the adapted software and is intimate with it does not have to undertake the understanding effort. See the Cost Factor Appendix.

Total Equivalent Size

The nonlinear effects for SU and UNFM are added to the linear approximation given by AAF to compute ESLOC. The overall Adaptation Adjustment Multiplier (AAM) is given by:

$$AAM = AAF(1 + .02 \cdot SU \cdot UNFM), \; AAF \leq 50\%$$
$$AAM = AAF + SU \cdot UNFM, \; AAF > 50\%$$

The total equivalent size for software composed of new and adapted software is:

$$Total\ Equivalent\ Size = New\ Size + AAM \cdot Adapted\ Size$$

Code counting tools can be used to measure CM. See the section on the Unified Code Count (UCC) tool, its capabilities, sample output and access to it.

Volatility

Volatility is requirements evolution and change, but not code thrown out. To account for the added effort, volatility is expressed as an additional percentage to size to obtain the total equivalent size for estimation.

Development Effort

Discuss:

- *Hours or Person Months*
- *Hours per Person Month discussion*
- *Labor categories to include*
- *Effort scope (Lifecycle phases covered)*
- *Impact of overlapping phases on collection*

Software development involves much more activity than just coding. It includes the work involved in developing requirements, designs and tests. It involves documentation and reviews, configuration management, and quality assurance. It can be done using different life cycles (agile, incremental, spiral, waterfall, etc.) and different ways of organizing the work (matrix, product lines, etc.). Using the DoD Software Resource Data Report as the basis, the following work activities/phases are included or excluded for effort.

Effort Activities and Phases

	Includes	Excludes
Activity		
System Conceptualization		✓
Systems Requirements Development		✓
Software Requirements Analysis	✓	
Software Architecture and Detailed Design	✓	
Software Coding and Unit Test	✓	
Software Integration and System/Software Integration	✓	
Hardware/Software Integration and Test		✓
System Test and Evaluation		✓
Operational Test and Evaluation		✓
Production		✓
Phase		
Inception	✓	
Elaboration	✓	
Construction	✓	
Transition		✓

Software requirements analysis includes any prototyping activities. The excluded activities are normally supported by software personnel but are considered outside the scope of their responsibility for effort measurement. Systems Requirements Development includes equations engineering (for derived requirements) and allocation to hardware and software.

All these activities include the effort involved in documenting, reviewing and managing the work-in-process. These include any prototyping and the conduct of demonstrations during the development. Once software is delivered to systems integration and test, it is viewed outside the scope of the software development team's responsibility.

Transition to operations and operations and support activities are not addressed by these analyses for the following reasons:

- They are normally accomplished by different organizations or teams.
- They are separately funded using different categories of money within the DoD.
- The cost data collected by projects therefore does not include them within their scope.

From a life cycle point-of-view, the activities comprising the software life cycle are represented for new, adapted, reused, generated and COTS (Commercial Off-The-Shelf) developments in Figure 2. Reconciling the effort associated with the activities in the Work Breakdown Structure (WBS) across life cycle is necessary for valid comparisons to be made between results from cost models. This Figure tries to show the different types of activities that need to be performed to integrate different types of components into the software builds and deliveries. For completeness, the following definitions are offered for each of these software types whose sizing in terms of equivalent new Source Lines of Code (SLOC) is covered in Chapter TBD of this Manual.

Schedule

Discuss:

- *Days, Months or Years*
- *Duration by phase*
- *Phase start and end criteria*
- *Impact of overlapping phases on collection*

References

Cost Estimation Models

Cost estimation models widely used on DoD projects are overviewed in this section. It describes the parametric software cost estimation model formulas, size inputs, cost drivers and how they relate to the standard Metrics Definitions. The models include COCOMO, SEER-SEM, SLIM, and True S. The similarities and differences for the cost model inputs (size, cost factors) and outputs (phases, activities) are identified for comparison.

Effort Formula

Parametric cost models used in avionics, space, ground, and shipboard platforms by the services are generally based on the common effort formula shown below. Size of the software is provided in a number of available units, cost factors describe the overall environment and calibrations may take the form of coefficients adjusted for actual data or other types of factors that account for domain-specific attributes [1] [2]. The total effort is calculated and then decomposed by phases or activities according to different schemes in the models.

$$Effort = A \cdot Size^{B} \cdot EAF$$

Where

- *Effort* is in person-months
- *A* is a calibrated constant
- *B* is a size scale factor
- *EAF* is an Effort Adjustment Factor from cost factor multipliers.

The popular parametric cost models in widespread use today allow size to be expressed as lines of code, function points, object-oriented metrics and other measures. Each model has its own respective cost factors and multipliers for *EAF*, and each model specifies the B scale factor in slightly different ways (either directly or through other factors). Some models use project type or application domain to improve estimating accuracy. Others use alternative mathematical formulas to compute their estimates. A comparative analysis of the cost models is provided next, including their sizing, WBS phases and activities.

Cost Models

The models covered include COCOMO II, SEER-SEM, SLIM, and True S. They were selected because they are the most frequently used models for estimating DoD software effort, cost and schedule. A comparison of the COCOMO II, SEER-SEM and True S models for NASA projects is described in [Madachy-Boehm 2008]. A previous study at JPL analyzed the same three models with respect to some of their flight and ground projects [Lum et al. 2001]. The consensus of these studies is any of the models can be used effectively if it is calibrated properly. Each of the models has strengths and each has weaknesses. For this reason, the studies recommend using at least two models to estimate costs whenever it is possible to provide added assurance that you are within an acceptable range of variation.

Other industry cost models such as SLIM, Checkpoint and Estimacs have not been as frequently used for defense applications as they are more oriented towards business applications per [Madachy-Boehm 2008]. A previous comparative survey of software cost models can also be found in [Boehm et al. 2000b]. COCOMO II is a public domain model that USC continually updates and is implemented in several commercial tools. True S and SEER-SEM are both proprietary commercial tools with unique features but also share some aspects with COCOMO. All three have been extensively used and tailored for flight project domains. SLIM is another parametric tool that uses a different approach to effort and schedule estimation.

COCOMO II

The COCOMO (COnstructive COst MOdel) cost and schedule estimation model was originally published in [3]. COCOMO II research started in 1994, and the model continues to be updated at USC with the rest of the COCOMO model family. COCOMO II defined in [Boehm et al. 2000] has three submodels: Applications Composition, Early Design and Post-Architecture. They can be combined in various ways to deal with different software environments. The Application Composition model is used to estimate effort and schedule on projects typically done as rapid application development. The Early Design model involves the exploration of alternative system architectures and concepts of operation. This model is based on function points (or lines of code when available) and a set of five scale factors and seven effort multipliers.

The Post-Architecture model is used when top level design is complete and detailed information about the project is available and the

software architecture is well defined. It uses Source Lines of Code and/or Function Points for the sizing parameter, adjusted for reuse and breakage; a set of 17 effort multipliers and a set of five scale factors that determine the economies/diseconomies of scale of the software under development. This model is the most frequent mode of estimation and used throughout this manual. The effort formula is:

$$Effort = A\ Size^{B} \prod_{i=1}^{N} EM_i$$

Where

- *Effort* is in person-months
- *A* is a constant derived from historical project data
- *Size* is in KSLOC (thousand source lines of code), or converted from other size measures
- *B* is an exponent for the diseconomy of scale dependent on additive scale drivers
- EM_i is an effort multiplier for the i^{th} cost driver. The geometric product of N multipliers is an overall effort adjustment factor to the nominal effort.

The COCOMO II effort is decomposed by lifecycle phase and activity as detailed in following model comparisons. More information on COCOMO can be found at http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html. A web-based tool for the model is at http://csse.usc.edu/tools/COCOMO.

SEER-SEM

SEER-SEM is a product offered by Galorath, Inc. This model is based on the original Jensen model [Jensen 1983], and has been on the market over 15 years. The Jensen model derives from COCOMO and other models in its mathematical formulation. However, its parametric modeling equations are proprietary. Like True S, SEER-SEM estimates can be used as part of a composite modeling system for hardware/software systems. Descriptive material about the model can be found in [Galorath-Evans 2006].

The scope of the model covers all phases of the project life-cycle, from early specification through design, development, delivery and maintenance. It handles a variety of environmental and application configurations, and models different development methods and languages. Development modes covered include object oriented, reuse, COTS, spiral, waterfall, prototype and incremental development. Languages covered are 3rd and 4th generation languages (C++, FORTRAN, COBOL, Ada, etc.), as well as application generators.

The SEER-SEM cost model allows probability levels of estimates, constraints on staffing, effort or schedule, and it builds estimates upon a knowledge base of existing projects. Estimate outputs include effort, cost, schedule, staffing, and defects. Sensitivity analysis is also provided as is a risk analysis capability. Many sizing methods are available including lines of code and function points. For more information, see the Galorath Inc. website at http://www.galorath.com.

SLIM

The SLIM model developed by Putnam is based on a Norden/Rayleigh manpower distribution and his finding in analyzing many completed projects [Putnam and Myers 1992]. The central part of Putnam's model called the software equation is:

$$S = C_k \cdot Effort^{\frac{1}{3}} \cdot t_d^{\frac{4}{3}}$$

Where

- *Effort* is in person-months
- *S* is the software delivery time
- C_k is a productivity environment factor.

The productivity environment factor reflects the development capability derived from historical data using the software equation. The size S is in LOC and the Effort is in person-years. Another relation found by Putnam is

(4) where D0 is the manpower build-up parameter which ranges from 8 (entirely new software with many interfaces) to 27 (rebuilt software). Combining the above equation with the software equation, we obtain the power function forms for effort and schedule:

.

(5)

.

(6) Putnam's model is used in the SLIM software tool based on this model for cost estimation and manpower scheduling [QSM 2003].

SLIM relies on the Rayleigh-Norden curve. The shape of the curve is determined by three key parameters within the model: size, PI, and the manpower buildup index (MBI). The MBI is a number that varies between -3 and 10. It reflects the rate at which personnel are added to a project. Higher ratings indicate faster buildup rates, and result in shorter schedules but higher costs. The Rayleigh curve is shifted upward and to the left as MBI increases. Lower size or higher PI values result in both lower costs and shorter schedules; the Rayleigh curve is shifted downward and to the left. Although MBI is a significant parameter, the user cannot input MBI directly. Instead, it is determined primarily by the user-specified constraints.

Lifecycle Coverage

True S

True S is the updated product to the PRICE S model offered by PRICE Systems. PRICE S was originally developed at RCA for use internally on software projects such as the Apollo moon program, and was then released in 1977 as a proprietary model. It fits into a composite modeling system and can be used to estimate more than just software costs. Many of the model's central algorithms were published in [Park 1988]. For more details on the model and the modeling system see the PRICE Systems website at http://www.pricesystems.com.

The PRICE S model consists of three submodels that enable estimating costs and schedules for the development and support of computer systems. The model covers business systems, communications, command and control, avionics, and space systems. PRICE S includes features for reengineering, code generation, spiral development, rapid development, rapid prototyping, object-oriented development, and software productivity measurement. Size inputs include SLOC, function points and/or Predictive Object Points (POPs). The True S system also provides a COCOMO II capability.

The switch to True S is recent. Hence some of the descriptions retain the old PRICE S terminology (such as the Rosetta Stone) while we move towards a complete True S implementation. Numeric estimate results shown are for the latest True S model.

The TruePlanning estimation suite from PRICE Systems contains both the True S model and the COCOMO II cost model.

Model Comparisons

Comparisons between the models for the core metric definitions of size, activities and lifecycle phases follow.

Size Inputs

This section describes the major similarities and differences between the models related to software sizing. All models support size inputs for new and adapted software, and some support automatically translated or generated code. The models differ with respect to their detailed parameters for the developed categories of software per below.

COCOMO II Size Inputs	SEER-SEM Size Inputs	True S Size Inputs
New Software		
New Size	New Size	New Size New Size Non-executable
Modified Software		
Adapted Size % Design Modified (DM) % Code Modified (CM) % Integration Required (IM)	Pre-exists Size [1] Deleted Size Redesign Required % Reimplementation Required %	Adapted Size Adapted Size Non-executable Amount of Modification % of Design Adapted

Assessment and Assimilation (AA) Software Understanding (SU) Programmer Unfamiliarity (UNFM)	Retest Required %	% of Code Adapted % of Test Adapted Deleted Size Code Removal Complexity
Reused Software		
Reused Size % Integration Required (IM) Assessment and Assimilation (AA)	Pre-exists Size [1, 2] Deleted Size Redesign Required % Reimplementation Required % Retest Required %	Reused Size [2] Reused Size Non-executable % of Design Adapted % of Code Adapted % of Test Adapted Deleted Size Code Removal Complexity
Generated Code		
		Auto Generated Code Size Auto Generated Size Non-executable
Automatically Translated		
Adapted SLOC Automatic Translation Productivity % of Code Reengineered		Auto Translated Code Size Auto Translated Size Non-executable
Deleted Code		
Volatility		
Requirements Evolution and Volatility (REVL)	Requirements Volatility (Change) [3]	TBD

1 - Specified separately for Designed for Reuse and Not Designed for Reuse

2 - Reused is not consistent with AFCAA definition if DM or CM >0

3 - Not a size input but a multiplicative cost driver

Units

The primary unit of software size in the effort models is Thousands of Source Lines of Code (KSLOC). KSLOC can be converted from other size measures, and additional size units can be used directly in the models as described next. User-defined proxy sizes can be developed for any of the models.

COCOMO II

The COCOMO II size model is based on SLOC or function points converted to SLOC, and can be calibrated and used with other software size units. Examples include use cases, use case points, object points, physical lines, and others. Alternative size measures can be converted to lines of code and used directly in the model or it can be independently calibrated directly to different measures.

SEER-SEM

Several sizing units can be used alone or in combination. SEER can use SLOC, function points and custom proxies. COTS elements are sized with *Features* and *Quick Size*. SEER allows proxies as a flexible way to estimate software size. Any countable artifact can be established as measure. Custom proxies can be used with other size measures in a project. Available pre-defined proxies that come with SEER include *Web Site Development*, *Mark II Function Point, Function Points* (for direct IFPUG-standard function points) and *Object-Oriented Sizing*.

SEER converts all size data into internal size units, also called effort units, Sizing in SEER-SEM can be based on function points, source lines of code, or user-defined metrics. Users can combine or select a single metric for any project element or for the entire project. COTS WBS elements also have specific size inputs defined either by Features, Object Sizing, or Quick Size, which describe the functionality being integrated.

New Lines of Code are the original lines created for the first time from scratch.

Pre-Existing software is that which is modified to fit into a new system. There are two categories of pre-existing software:

- *Pre-existing, Designed for Reuse*

- *Pre-existing, Not Designed for Reuse.*

Both categories of pre-existing code then have the following subcategories:

Pre-existing lines of code which is the number of lines from a previous system

Lines to be Deleted are those lines deleted from a previous system.

Redesign Required is the percentage of existing code that must be redesigned to meet new system requirements.

Reimplementation Required is the percentage of existing code that must be re-implemented, physically recoded, or reentered into the system, such as code that will be translated into another language.

Retest Required is the percentage of existing code that must be retested to ensure that it is functioning properly in the new system.

SEER then uses different proportional weights with these parameters in their AAF equation according to

*Pre-existing Effective Size = (0.4*A + 0.25*B + 0.35*C).*

Where A, B and C are the respective percentages of code redesign, code reimplementation, and code retest required.

Function-Based Sizing

- External Input (EI)
- External Output (EO)
- Internal Logical File (ILF)
- External Interface Files (EIF)
- External Inquiry (EQ)
- Internal Functions (IF) ,Äì any functions that are neither data nor transactions

Proxies

- Web Site Development
- Mark II Function Points
- Function Points (direct)
- Object-Oriented Sizing.

COTS Elements

- Quick Size
- Application Type Parameter
- Functionality Required Parameter
- Features
- Number of Features Used
- Unique Functions
- Data Tables Referenced
- Data Tables Configured

True S

The True S software cost model size measures may be expressed in different size units including source lines of code (SLOC), function points, predictive object points (POPs) or Use Case Conversion Points (UCCPs). True S also differentiates executable from non-executable software sizes. *Functional Size* describes software size in terms of the functional requirements that you expect a Software COTS component to satisfy. Also see section TBD. The True S software cost model size definitions for all of the size units are listed below.

Adapted Code Size - This describes the amount of existing code that must be changed, deleted, or adapted for use in the new software project. When the value is zero (0.00), the value for New Code Size or Reused Code Size must be greater than zero.

Adapted Size Non-executable - This value represents the percentage of the adapted code size that is non-executable (such as data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. When a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Amount for Modification - This represents the percent of the component functionality that you plan to modify, if any. The Amount for Modification value (like Glue Code Size) affects the effort calculated for the Software Design, Code and Unit Test, Perform Software Integration and Test, and Perform Software Qualification Test activities.

Auto Gen Size Non-executable - This value represents the percentage of the Auto Generated Code Size that is non-executable (such as, data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Auto Generated Code Size - This value describes the amount of code generated by an automated design tool for inclusion in this component.

Auto Trans Size Non-executable - This value represents the percentage of the Auto Translated Code Size that is non-executable (such as, data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Auto Translated Code Size - This value describes the amount of code translated from one programming language to another by using an automated translation tool (for inclusion in this component).

Auto Translation Tool Efficiency - This value represents the percentage of code translation that is actually accomplished by the tool. More efficient auto translation tools require more time to configure the tool to translate. Less efficient tools require more time for code and unit test on code that is not translated.

Code Removal Complexity - This value describes the difficulty of deleting code from the adapted code. Two things need to be considered when deleting code from an application or component: the amount of functionality being removed and how tightly or loosely this functionality is coupled with the rest of the system. Even if a large amount of functionality is being removed, if it accessed through a single point rather than from many points, the complexity of the integration will be reduced.

Deleted Code Size - This describes the amount of pre-existing code that you plan to remove from the adapted code during the software project. The Deleted Code Size value represents code that is included in Adapted Code Size, therefore, it must be less than, or equal to, the Adapted Code Size value.

Equivalent Source Lines of Code - The ESLOC (Equivalent Source Lines of Code) value describes the magnitude of a selected cost object in Equivalent Source Lines of Code size units. True S does not use ESLOC in routine model calculations, but provides an ESLOC value for any selected cost object. Different organizations use different formulas to calculate ESLOC.

The True S calculation for ESLOC is

$$ESLOC = NewCode + .70(AdaptedCode) + .10(ReusedCode).$$

To calculate ESLOC for a Software COTS, True S first converts Functional Size and Glue Code Size inputs to SLOC using a default set of conversion rates. NewCode includes Glue Code Size and Functional Size when the value of Amount for Modification is greater than or equal to 25%. AdaptedCode includes Functional Size when the value of Amount for Modification is less than 25% and greater than zero. ReusedCode includes Functional Size when the value of Amount for Modification equals zero.

Functional Size - This value describes software size in terms of the functional requirements that you expect a Software COTS component to satisfy. When you select Functional Size as the unit of measure (Size Units value) to describe a Software COTS component, the Functional Size value represents a conceptual level size that is based on the functional categories of the software (such as Mathematical, Data Processing, or Operating System). A measure of Functional Size can also be specified using Source Lines of Code, Function Points, Predictive Object Points or Use Case Conversion Points if one of these is the Size Unit selected.

Glue Code Size - This value represents the amount of glue code that will be written. Glue Code holds the system together, provides interfaces between Software COTS components, interprets return codes, and translates data into the proper format. Also, Glue Code may be required to compensate for inadequacies or errors in the COTS component selected to deliver desired functionality.

New Code Size - This value describes the amount of entirely new code that does not reuse any design, code, or test artifacts. When the value is zero (0.00), the value must be greater than zero for Reused Code Size or Adapted Code Size.

New Size Non-executable - This value describes the percentage of the New Code Size that is non-executable (such as data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.0 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Percent of Code Adapted - This represents the percentage of the adapted code that must change to enable the adapted code to function and meet the software project requirements.

Percent of Design Adapted - This represents the percentage of the existing (adapted code) design that must change to enable the adapted code to function and meet the software project requirements. This value describes the planned redesign of adapted code. Redesign includes architectural design changes, detailed design changes, and any necessary reverse engineering.

Percent of Test Adapted - This represents the percentage of the adapted code test artifacts that must change. Test plans and other artifacts must change to ensure that software that contains adapted code meets the performance specifications of the Software Component cost object.

Reused Code Size - This value describes the amount of pre-existing, functional code that requires no design or implementation changes to function in the new software project. When the value is zero (0.00), the value must be greater than zero for New Code Size or Adapted Code Size.

Reused Size Non-executable - This value represents the percentage of the Reused Code Size that is non-executable (such as, data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

SLIM

SLIM uses effective system size composed of new, modified and reused code. Deleted code is not considered in the model. If there is reused code than the Productivity Index (PI) factor may be adjusted to add in time and effort for regression testing and integration of the reused code.

SLIM provides different sizing techniques including:

- Sizing by history
- Total system mapping
- Sizing by decomposition
- Sizing by module
- Function point sizing.

Alternative sizes to SLOC such as use cases or requirements can be used in Total System Mapping. The user defines the method and quantitative mapping factor.

Lifecycles, Activities and Cost Categories

COCOMO II allows effort and schedule to be allocated to either a waterfall or MBASE lifecycle. MBASE is a modern iterative and incremental lifecycle model like the Rational Unified Process (RUP) or the Incremental Commitment Model (ICM). The phases include: (1) Inception, (2) Elaboration, (3) Construction, and (4) Transition.

True-S uses the nine DoD-STD-2167A development phases: (1) Concept, (2) System Requirements, (3) Software Requirements, (4) Preliminary Design, (5) Detailed Design, (6) Code/Unit test, (7) Integration & Test, (8) Hardware/Software Integration, and (9) Field Test.

In SEER-SEM the standard lifecycle activities include: (1) System Concept, (2) System Requirements Design, (3) Software Requirements Analysis, (4) Preliminary Design, (5) Detailed Design, (6) Code and Unit Test, (7) Component Integration and Testing, (8) Program Test, (9) Systems Integration through OT&E & installation, and (10) Operation Support. Activities may be defined differently across development organizations and mapped to SEER-SEMs designations.

In SLIM the lifecycle maps to four general phases of software development. The default phases are: 1) Concept Definition, 2) Requirements and Design, 3) Construct and Test, and 4) Perfective Maintenance. The phase names, activity descriptions and deliverables can be changed in SLIM. The “main build” phase initially computed by SLIM includes the detailed design through system test phases, but the model has the option to include the “requirements and design” phase, including software requirements and preliminary design, and a “feasibility study” phase to encompass system requirements and design.

The phases covered in the models are summarized in the next table.

Model	Phases
COCOMO II	■ Inception ■ Elaboration

	■ Construction ■ Transition ■ Maintenance
SEER-SEM	■ System Requirements Design ■ Software Requirements Analysis ■ Preliminary Design ■ Detailed Design ■ Code / Unit Test ■ Component Integrate and Test ■ Program Test ■ System Integration Through OT&E
True S	■ Concept ■ System Requirements ■ Software Requirements ■ Preliminary Design ■ Detailed Design ■ Code / Unit Test ■ Integration and Test ■ Hardware / Software Integration ■ Field Test ■ System Integration and Test ■ Maintenance
SLIM	■ Concept Definition ■ Requirements and Design ■ Construction and Testing ■ Maintenance

The work activities estimated in the respective tools are in the next table.

Model	Activities
COCOMO II	■ Management ■ Environment / CM ■ Requirements ■ Design ■ Implementation ■ Assessment ■ Deployment
SEER-SEM	■ Management ■ Software Requirements ■ Design ■ Code ■ Data Programming ■ Test ■ CM ■ QA
True S	■ Design ■ Programming ■ Data ■ SEPGM ■ QA ■ CFM

SLIM	■ WBS Sub-elements of Phases: ■ Concept Definition ■ Requirements and Design ■ Construct and Test ■ Perfective Maintenance

The categories of categories covered in the estimation models and tools are listed in the next table.

Model	Categories
COCOMO II	■ Software Engineering Labor
SEER-SEM	■ Software Engineering Labor ■ Purchases
True S	■ Software Engineering Labor ■ Purchased Good ■ Purchased Service ■ Other Cost
SLIM	■ Software Engineering Labor

References

1. ↑ Lum K, Powell J, Hihn J, “Validation of Spacecraft Software Cost Estimation Models for Flight and Ground Systems”, JPL Report, 2001
2. ↑ Madachy R, Boehm B, “Comparative Analysis of COCOMO II, SEER-SEM and True-S Software Cost Models”, USC-CSSE-2008-816, 2008
3. ↑ Boehm B., Software Engineering Economics. Englewood Cliffs, NJ, Prentice-Hall, 1981

SRDR Process

The Software Resources Data Report (SRDR) is used to obtain both the estimated and actual characteristics of new software developments or upgrades. Both the Government program office and, later on after contract award, the software contractor submit this report. For contractors, this report constitutes a contract data deliverable that formalizes the reporting of software metric and resource data. All contractors, developing or producing any software development element with a projected software effort greater than $20M (then year dollars) on major contracts and subcontracts within ACAT I and ACAT IA programs, regardless of contract type, must submit SRDRs. The data collection and reporting applies to developments and upgrades whether performed under a commercial contract or internally by a government Central Design Activity (CDA) under the terms of a memorandum of understanding (MOU).

Access DCARC database

Defense Cost and Resource Center (DCARC) and its Web site, http://dcarc.pae.osd.mil.

The DCARC, which is part of OSD Cost Assessment and Program Evaluation (CAPE), exists to collect Major Defense Acquisition Program (MDAP) cost and software resource data and make those data available to authorized Government analysts. This website is the authoritative source of information associated with the Cost and Software Data Reporting (CSDR) system, including but not limited to: policy and guidance, training materials, and data. CSDRs are DoD’s only systematic mechanism for capturing completed development and production contract "actuals" that provide the right visibility and consistency needed to develop credible cost estimates. Since credible cost estimates enable realistic budgets, executable contracts and program stability, CSDRs are an invaluable resource to the DoD cost analysis community and the entire DoD acquisition community.

The Defense Cost and Resource Center (DCARC), was established in 1998 to assist in the re-engineering of the CCDR process. The DCARC is part of OSD Cost Assessment and Program Evaluation (CAPE). The primary role of the DCARC is to collect current and historical Major Defense Acquisition Program cost and software resource data in a joint service environment and make those data available for use by authorized government analysts to estimate the cost of ongoing and future government programs, particularly DoD weapon systems.

The DCARC's Defense Automated Cost Information Management System (DACIMS) is the database for access to current and historical cost and software resource data needed to develop independent, substantiated estimates. DACIMS is a secure website that allows DoD government cost estimators and analysts to browse through almost 30,000 CCDRs, SRDR and associated documents via the Internet. It is the largest repository of DoD cost information. Use analysis procedures in this manual to interpret the SRDR data

Content

The SRDR Final Developer Report contains actual, as-built software measurement data as described in the contractor's SRDR Data Dictionary. The data shall reflect scope relevant to the reporting event.

- SRDR submissions for contract complete event shall reflect the entire software development project.
- When the development project is divided into multiple product builds, each representing production level software delivered to the government, the submission should reflect each product build.
- SRDR submissions for completion of a product build shall reflect size, effort, and schedule of that product build.

The report consists of two pages. The fields in each page are briefly described below.

- Page 1
- Page 2

Administrative Information (SRDR Section 3.1)

Product and Development Description (SRDR Section 3.2)

- Functional Description. A brief description of its function.
- Software Development Characterization
- Application Type
  - Primary and Secondary Programming Language.
  - Percent of Overall Product Size. Approximate percentage (up to 100%) of the product size that is of this application type.
  - Actual Development Process. Enter the name of the development process followed for the development of the system.
  - Software Development Method(s). Identify the software development method or methods used to design and develop the software product .
  - Upgrade or New Development? Indicate whether the primary development was new software or an upgrade.
  - Software Reuse. Identify by name and briefly describe software products reused from prior development efforts (e.g. source code, software designs, requirements documentation, etc.).
- COTS/GOTS Applications Used.
  - Name. List the names of the applications or products that constitute part of the final delivered product, whether they are COTS, GOTS, or open-source products.
  - Integration Effort (Optional). If requested by the CWIPT, the SRD report shall contain the

actual effort required to integrate each COTS/GOTS application identified in Section 3.2.4.1.

- Staffing.
  - Peak Staff. The actual peak team size, measured in full-time equivalent (FTE) staff.
  - Peak Staff Date. Enter the date when the actual peak staffing occurred.
  - Hours per Staff-Month. Enter the number of direct labor hours per staff-month.
- Personnel Experience in Domain. Stratify the project staff domain experience by experience level and specify the percentage of project staff at each experience level identified. Sample Format 3 identifies five levels:
  - Very Highly Experienced (12 or more years)
  - Highly Experienced (6 to 12 years)
  - Nominally Experienced (3 to 6 years)
  - Low Experience (1 to 3 years)
  - Inexperienced/Entry Level (less than a year).

Product Size Reporting (SRDR Section 3.3)

- Number of Software Requirements. Provide the actual number of software requirements.
  - Total Requirements. Enter the actual number of total requirements satisfied by the developed software product at the completion of the increment or project.
  - New Requirements. Of the total actual number of requirements reported, identify how many are new requirements.
- Number of External Interface Requirements. Provide the number of external interface requirements, as specified below, not under project control that the developed system satisfies.
  - Total External Interface Requirements. Enter the actual number of total external interface requirements satisfied by the developed software product at the completion of the increment or project.
  - New External Interface Requirements. Of the total number of external interface requirements reported, identify how many are new external interface requirements.
- Requirements Volatility. Indicate the amount of requirements volatility encountered during development as a percentage of requirements that changed since the Software Requirements Review.
- Software Size.
  - Delivered Size. Capture the delivered size of the product developed, not including any code that was needed to assist development but was not delivered (such as temporary stubs, test scaffoldings, or debug statements). Additionally, the code shall be partitioned (exhaustive with no overlaps) into appropriate development categories. A common set of software development categories is new, reused with modification, reused without modification, carry-over code, deleted code, and auto-generated code.
    - Reused Code With Modification. When code is included that was reused with modification, provide an assessment of the amount of redesign, recode, and retest required to implement the modified or reused code.
    - Reuse Code Without Modification. Code reused without modification is code that has no design or code modifications. However, there may be an amount of retest required. Percentage of retest should be reported with the retest factors described above.
    - Carryover Code. Report shall distinguish between code developed in previous increments that is carried forward into the current increment and code added as part of the effort on the current increment.

        - Deleted Code. Include the amount of delivered code that was created and subsequently deleted from the final delivered code
        - Auto-generated Code. If the developed software contains auto-generated source code, report an auto-generated code sizing partition as part of the set of development categories.
        - Subcontractor-Developed Code.
    - Counting Convention. Identify the counting convention used to count software size.
    - Size Reporting by Programming Language (Optional).
    - Standardized Code Counting (Optional). If requested, the contractor shall use a publicly available and documented code counting tool, such as the University of Southern California Code Count tool, to obtain a set of standardized code counts that reflect logical size. These results shall be used to report software sizing.

Resource and Schedule Reporting (SRDR Section 3.4)

The Final Developer Report shall contain actual schedules and actual total effort for each software development activity.

- Effort. The units of measure for software development effort shall be reported in staff-hours. Effort shall be partitioned into discrete software development activities .
- WBS Mapping.
- Subcontractor Development Effort. The effort data in the SRD report shall be separated into a minimum of two discrete categories and reported separately: Prime Contractor Only and All Other Subcontractors.
- Schedule. For each software development activity reported, provide the actual start and end dates for that activity.

Product Quality Reporting (SRDR Section 3.5 - Optional)

Quality should be quantified operationally (through failure rate and defect discovery rate). However, other methods may be used if appropriately explained in the associated SRDR Data Dictionary.* Defects

- Number of Defects Discovered. Report an estimated number of defects discovered during integration and qualification testing. If available, list the expected defect discovery counts by priority, e.g. 1, 2, 3, 4, 5. Provide a description of the priority levels if used.
- Number of Defects Removed. Report an estimated number of defects removed during integration and qualification testing. If available, list the defect removal counts by priority.

SRDR Data Dictionary

The SRDR Data Dictionary shall contain, at a minimum, the following information in addition to the specific requirements identified in Sections 3.1 through 3.5:

- Experience Levels. Provide the contractor's specific definition (i.e., the number of years of experience) for personnel experience levels reported in the SRD report.
- Software Size Definitions. Provide the contractor's specific internal rules used to count software code size.
- Software Size Categories. For each software size category identified (i.e., New, Modified, Unmodified, etc.), provide the contractor's specific rules and/or tools used for classifying code into each category.
- Peak Staffing. Provide a definition that describes what activities were included in peak staffing.
- Requirements Count (Internal). Provide the contractor's specific rules and/or tools used to count

requirements.
- Requirements Count (External). Provide the contractor's specific rules and/or tools used to count external interface requirements.
- Requirements Volatility. Provide the contractor's internal definitions used for classifying requirements volatility.
- Software Development Activities. Provide the contractor's internal definitions of labor categories and activities included in the SRD report's software activity.
- Product Quality Reporting. Provide the contractor's internal definitions for product quality metrics being reported and specific rules and/or tools used to count the metrics.

References

Domain Analysis

Cost and schedule estimating relationships (CER and SER) are different for different types of software. Factors such as application complexity, impact of loss due to reliability, autonomous modes of operation, constraints on timing, storage and power, security requirements, and complex interfaces influence the cost and time to develop applications. Parametric cost models have a number of adjustable parameters that attempt to account for these factors.

Application Domain Decomposition

Instead of developing CERs and SERs with many parameters, this chapter describes an analysis approach based on grouping similar software applications together. These groups are called **Domains**. Domains implement a combination of hardware and software components to achieve the intended functionality. Instead of using a domain name such as Communications, a better approach is to use a generic productivity type (PT). Also consideration needs to be given to the operating environment that the domain operates within. Both the operating environment and domain are considered in this analysis to produce the productivity types.

Operating Environments

Operating Environments have similar systems, similar products, similar operational characteristics, and similar requirements:

- High-speed vehicle versus stationary
- Battery operated versus ground power
- Unrecoverable platform versus readily accessible
- Limited, non-upgradeable computing processor capacity versus racks of processors
- Fixed internal and external memory capacity versus expandable capacity

There are 9 operating environments:

Operating Environment	Examples
Manned Ground Site (MGS)	Command Post, Ground Operations Center, Ground Terminal, Tracking Station, Dispatch Center, Test Facilities
Ground Surface Vehicles	
• Manned Ground Vehicle (MGV)	Tanks
• Unmanned Ground Vehicle (UGV)	Robots

Maritime Systems • Manned Maritime Vessel (MMV) • Unmanned Maritime Vessel (UMV)	 Aircraft carriers, destroyers, supply ships, submarines Mine hunting systems
Aerial • Manned Aerial Vehicle (MAV) • Unmanned Aerial Vehicle (UAV)	 Fixed-wing aircraft, Helicopters Remotely piloted air vehicles
Unmanned Ordinance Vehicle (UOV)	Both powered and unpowered ordinance: Air-to-air missiles, Air-to-ground missiles, Smart bombs, Ground-to-ground missiles
Space • Manned Space Vehicle (MSV) • Unmanned Space Vehicle (USV)	 Space shuttle, Space passenger vehicle, Manned space stations Orbiting satellites (weather, communications), Exploratory space vehicles

Productivity Types

Productivity types are groups of application productivities that are environment independent, technology driven, and are characterized by the following:

- Required software reliability
- Database size - if there is a large data processing and storage component to the software application
- Product complexity
- Integration complexity
- Real-time operating requirements
- Platform volatility, Target system volatility
- Special display requirements
- Development re-hosting
- Quality assurance requirements
- Security requirements
- Assurance requirements
- Required testing level

There are 14 productivity types:

Productivity Types	Description
Sensor Control and Signal Processing (SCP)	Software that requires timing-dependent device coding to enhance, transform, filter, convert, or compress data signals. Ex.: Bean steering controller, sensor receiver/transmitter control, sensor signal processing, sensor receiver/transmitter test. Ex. of sensors: antennas, lasers, radar, sonar, acoustic, electromagnetic.
Vehicle Control (VC)	Hardware & software necessary for the control of vehicle primary and secondary mechanical devices and surfaces. Ex: Digital Flight Control, Operational Flight Programs, Fly-By-Wire Flight Control System, Flight Software, Executive.

Real Time Embedded (RTE)	Real-time data processing unit responsible for directing and processing sensor input/output. Ex: Devices such as Radio, Navigation, Guidance, Identification, Communication, Controls And Displays, Data Links, Safety, Target Data Extractor, Digital Measurement Receiver, Sensor Analysis, Flight Termination, Surveillance, Electronic Countermeasures, Terrain Awareness And Warning, Telemetry, Remote Control.
Vehicle Payload (VP)	Hardware & software which controls and monitors vehicle payloads and provides communications to other vehicle subsystems and payloads. Ex: Weapons delivery and control, Fire Control, Airborne Electronic Attack subsystem controller, Stores and Self-Defense program, Mine Warfare Mission Package.
Mission Processing (MP)	Vehicle onboard master data processing unit(s) responsible for coordinating and directing the major mission systems. Ex.: Mission Computer Processing, Avionics, Data Formatting, Air Vehicle Software, Launcher Software, Tactical Data Systems, Data Control And Distribution, Mission Processing, Emergency Systems, Launch and Recovery System, Environmental Control System, Anchoring, Mooring and Towing.
Command & Control (C&C)	Complex of hardware and software components that allow humans to manage a dynamic situation and respond to user-input in real time. Ex: Battle Management, Mission Control.
System Software (SYS)	Layers of software that sit between the computing platform and applications Ex: Health Management, Link 16, Information Assurance, Framework, Operating System Augmentation, Middleware, Operating Systems.
Telecommunications (TEL)	Transmission and receipt of voice, digital, and video data on different mediums & across complex networks. Ex: Network Operations, Communication Transport.
Process Control (PC)	Software that controls an automated system, generally sensor driven. Ex:
Scientific Systems (SCI)	Non real time software that involves significant computations and scientific analysis. Ex: Environment Simulations, Offline Data Analysis, Vehicle Control Simulators
Training (TRN)	Hardware and software that are used for educational and training purposes Ex: Onboard or Deliverable Training Equipment & Software, Computer-Based Training.
Test Software (TST)	Hardware & Software necessary to operate and maintain systems and subsystems which are not consumed during the testing phase and are not allocated to a specific phase of testing. Ex: Onboard or Deliverable Test Equipment & Software.
Software Tools (TUL)	Software that is used for analysis, design, construction, or testing of computer programs Ex: Integrated collection of tools for most development phases of the life cycle, e.g. Rational development environment
Business Systems (BIS)	Software that automates a common business function Ex: Database, Data Distribution, Information Processing, Internet, Entertainment, Enterprise Services*, Enterprise Information**

- *Enterprise Information: HW & SWneeded for developing functionality or software service that are unassociated, loosely coupled units of functionality.
  Examples are: Enterprise service management (monitoring, fault management), Machine-to-machine messaging, Service discovery, People and device discovery, Metadata discovery, Mediation, Service security, Content discovery and delivery, Federated search, Enterprise catalog service, Data source integration, Enterprise content delivery network (caching specification, distributed caching, forward staging), Session management,, Audio & video over internet protocol, Text collaboration (chat, instant messaging), Collaboration (white boarding & annotation), Application broadcasting and sharing, Virtual spaces, Identity management (people and device discovery), User profiling and customization.
- **Enterprise Information: HW & SW needed for assessing and tailoring COTS software applications or modules that can be attributed to a specific software service or bundle of

services.
Examples of enterprise information systems include but not limited to: , Enterprise resource planning, Enterprise data warehouse, Data mart, Operational data store.
Examples of business/functional areas include but not limited to: General ledger, Accounts payable, Revenue and accounts receivable, Funds control and budgetary accounting, Cost management, Financial reporting, Real property inventory and management.

Finding the Productivity Type

It can be challenging to determine which productivity type should be used to estimate the cost and schedule of an application (that part of the hardware-software complex which comprise a domain). The productivity types are by design generic. By using a work breakdown structure (WBS), the environment and domain are used to determine the productivity type.

Using the WBS from MIL-STD-881C, a mapping is created from environment to productivity type (PT). Starting with the environment, traverse the WBS to the lowest level where the domain is represented. Each domain is associated with a productivity type (PT). In real-world WBSs, the traverse from environment to PT will most likely not be the same number of levels. However the 881C WBS provides the context for selecting the PT which should be transferable to other WBSs.

Two examples of finding the PT using the 881C Manned Aerial Vehicle (MAV) and Unmanned Spacecraft (USC) WBS elements. The highest level element represents the environment. In the MAV environment there are the Avionics subsystem, Fire-Control sub-subsystem, and the sensor, navigation, air data, display, bombing computer and safety domains. Each domain has an associated productivity type.

Environment	Subsystem	Sub-subsystem	Domains	PT
MAV	Avionics	Fire Control	Search, target, tracking sensors	SCP
			Self-contained navigation	RTE
			Self-contained air data systems	RTE
			Displays, scopes, or sights	RTE
			Bombing computer	MP
			Safety devices	RTE
		Data Display and Controls	Multi-function display	RTE
			Control display units	RTE
			Display processors	MP
			On-board mission planning	TRN

For a space system, the highest level 881C WBS element is the Unmanned Spacecraft (USC). The two sub-systems are Bus and Payload. The domains for Bus address controlling the vehicle. The domains for Payload address controlling the onboard equipment. Each domain has an associated productivity type.

Environment	Subsystem	Domains	PT
USV	Bus	Structures & Mechanisms (SMS)	VC
		Thermal Control (TCS)	VC
		Electrical Power (EPS)	VC
		Attitude Control (ACS)	VC
		Propulsion	VC
		Telemetry, Tracking, & Command (TT&C)	RTE
		Bus Flight Software	VC
	Payload	Thermal Control	RTE
		Electrical Power	RTE
		Pointing, Command, & Control Interface	VP
		Payload Antenna	SCP
		Payload Signal Electronics	SCP
		Optical Assembly	SCP
		Sensor	SCP
		Payload Flight Software	VP

The full table is available for the MIL-STD-881C WBS Mapping to Productivity Types.

Analysis

Domain analysis of the SRDR database is presented in the next sections, and provides guidance in developing estimates in the respective domains. Cost and schedule estimating relationships are expressed in different forms. In this manual, they are expressed as a ratio commonly called Productivity and as a simple math equation called a Model.

Productivity-based CER

Software productivity refers to the ability of an organization to generate outputs using the resources that it currently has as inputs. Inputs typically include facilities, people, experience, processes, equipment, and tools. Outputs generated include software applications and documentation used to describe them.

The metric used to express software productivity is thousands of equivalent source lines of code (KESLOC) per person-month (PM) of effort. While many other measures exist, KESLOC/PM will be used because most of the data collected by the Department of Defense (DoD) on past projects is captured using these two measures. While controversy exists over whether or not KESLOC/PM is a good measure, consistent use of this metric (see Metric Definitions) provides for meaningful comparisons of productivity.

$$Productivity = \frac{Outputs}{Inputs} = \frac{KESLOC}{PM}$$

Productivity is not only influenced by the operating environment and domain type but size influences it as well. The larger the application being developed, the larger the number of overhead activities required to communicate and coordinate the development. Productivity decreases as size increases. For this reason, within an environment and domain, different productivities are shown for different groups of size:

- 0-25 KESLOC
- 26-50 KESLOC
- 51-100 KESLOC
- 100+ KESLOC

Analysis results are shown in the **Productivity-based CER** page.

Model-based CER & SER

Simple models with one or two parameters can be used to express cost and schedule estimating relationships. In this analysis, these models have the form:

CER:

$$PM = A * KESLOC^{B}$$

where estimated size, KESLOC, is used to predict effort, PM. A is a constant and B is a scaling factor that takes into account changing productivity with changing size.

SER:

$$DUR = A * KESLOC^{B} * AvgStaff^{C}$$

where estimated size, KESLOC, and estimated average staffing levels, AvgStaff, are used to predict effort, PM. A is a constant, B is a scaling factor to account for changing productivity as size increases, and C is a scaling factor to account for the non-linear relationship between changing staffing levels and shortening/lengthening duration, Dur.

Analysis results are shown in the **Model-based CER and SER** page.

References

Metrics Guidance

This section includes data normalization guidelines conforming to the Metrics Definitions, how to use the domain analysis results, default productivity metrics and cost model inputs, guidelines for estimating growth and assessing schedule, and overall lessons learned.

It examines rules for estimating software size based on available data (requirements, use cases, etc.). It incorporates historical and subjective factors for quantifying effective size based on domain and adaptation type. Guidance for different classes of software covers COTS and ERP projects. Software maintenance estimation is also addressed.

Detailed examples are provided throughout such as computing equivalent size in different scenarios and handling multiple increments.

Data Normalization

optional text

Line Counts

if using analogy to historical data based on physical SLOC, convert physical SLOC to logical SLOC.

Converting Line Counts to Logical Lines

Converting Line Counts

For analysis, the definition of a source line of code needs to be as consistent as possible to eliminate noise in the data. A logical source line of code has be selected as the baseline SLOC definition.

Three different types of SLOC counts were encountered in the SRDR data:

Total Count
: a line in a file, e.g. carriage returns including blanks and comment lines

Non-Commented Source Statements (NCSS) Count
: a line in a file that is not a blank or comment line

Logical Count
: as defined earlier [link to definition]

If a source line of code has been defined as either Total or NCSS, these counts need to be converted to a Logical SLOC count. An experiment was run using the UCC tool [link to UCC] on public domain software applications and additional contributions from USC-CSSE Affiliates. Total, NCSS and Logical counts were taken from the program files. Six programming languages were sampled: Ada, C#, C/C++, Java, PERL, and PHP. The number of data points was 40.

Total Line Count Conversion

The plot of the Total count to the Logical count shows a stable relationship. The relationship and a plot of the data are shown below:

Logical SLOC count = 0.33 * Total count

where the intercept was constrained to zero and

Adjusted R^2	0.98
Standard Error	0.01

Lower 90% Confidence Interval	0.32
Upper 90% Confidence Interval	0.34

[insert plot image here]

NCSS Line Count Conversion

Similiarly, the plot of the NCSS count to the Logical count shows a stable relationships. The relationship and a plot of the data are shown below:

Logical SLOC count = 0.64 * NCSS count

where the intercept was constrained to zero and

Adjusted R^2	0.99
Standard Error	0.01
Lower 90% Confidence Interval	0.62
Upper 90% Confidence Interval	0.65

[insert plot image here]

Optional use of checklists

Effort

Normalizing Effort to a fixed number of phasesTBD (Show descriptive statistics on Effort from data analysis)

Schedule

Normalizing Duration to a fixed number of phases

TBD (Show descriptive statistics on Duration from data analysis)

Example: New Software

A system is to be developed all new. There is no legacy software or other reusable software used. The only size input for estimation is New and there are no adaptation parameters involved.

Adapted Software Equivalent Size

The following examples show applications of quantifying adapted software to compute equivalent size with the previous equations (5) and (6).

Example: Modified Software

This example estimates the equivalent size associated with writing a new user interface to work with an existing application. Assume the size of the new interface is 20 KSLOC. For it to work, we must change the existing application to accommodate a new Application Program Interface (API). If the adapted size estimate is 100 KSLOC as follows under the assumption that the original code size was 8 KSLOC we compute:

AAM = [0.4 (5% DM) + 0.3 (10% CM) + 0.3 (10% IM)] [100KSLOC] = 8 KSLOC

Further assume that we are dealing with poorly-written spaghetti code and that we are totally unfamiliar with it. We would then rate SU as 50% and UNFM as 1. As a result, we would have to increase our estimate to reflect this learning curve.

Example: Upgrade to Legacy System

In this example there is a very large existing legacy system undergoing periodic upgrades. The size of the legacy system is so large that the equivalent size to estimate the incremental update is very sensitive to the adaptation parameters. The size metrics for the increment are:

- New code: 75 KSLOC
- Modified code: 20 KSLOC
- Legacy code: 3 MSLOC.

Care must be taken when assigning the adaptation parameters for the legacy code to compute its equivalent size. For example, the difference between the small values of AAF = 0% and AAF = 5% is a tripling of the equivalent size for the upgrade. Some regression testing of untouched legacy code is inevitable and the factor for % Integration Required should be investigated carefully in terms of the relative effort involved. This might be done by quantifying the number the regression tests performed and their manual intensity compared to the tests of new functionality. If the % Integration Required for the legacy code is 1% then the adaption factor for it would be:

AAM = [0.4 (0% DM) + 0.3 (0% CM) + 0.3 (1% IM)] * = .0003.

The total ESLOC for new, modified and reused (legacy) assuming the AAF for the modified code is 25% is:

ESLOC = 75 + 20*.25 + 3M * .003 = 75 + 5 + 9 = 89 KSLOC.

In this case, building on top of the legacy baseline was 9/89 or about 10% of the work.

Commercial Off-The-Shelf Software (COTS)

Sizing Issues The sizing of COTS software sizing is not addressed fully here since the source is not available to be modified. However, there are instances when size proxies are used for effort related to COTS in some models. Others can treat COTS as reused software or be used in conjunction with other COCOTS-specific models [Boehm et al. 2000, while others have more extensive COTS models built in.

When you estimate COTS without code counts, you must use another model based on the number of packages, standardized effort or other measures besides size in lines of code. Some common sources of COTS effort from the COCOTS model [Abts 2004] and methods to estimate them are shown in Table 21. Detailed tables for estimating these quantities in the COCOTS model are provided in [Boehm et al. 2000]. TBD (Glue code must be accounted for.)

The rationale for building systems with COTS components is that they will require less development time by taking advantage of existing, market proven, vendor supported products, thereby reducing overall system development costs. But the COTS product source code is not available to the application developer, and the future evolution of the COTS product is not under the control of the application developer. Thus there is a trade-off in using the COTS approach in that new software development time can be reduced, but generally at the cost of an increase in software component integration work.

Sometimes the adaptation parameters are relevant for purchased COTS software that must be configured, interfaced and tested without knowledge of internals (only have executables). Assume there is an API and glue code is treated as new code. SU and UNFM are used to gain enough insight to interface package via API. Assume there is an equivalent SLOC estimate derived via Function or Feature Points. In this case representative parameter settings for this case are shown in the row for Purchased software in Table 17.

Table 21: COTS Cost Source Estimation COTS Cost Source Description Cost Estimation Methods Assessment The process by which COTS components are selected for use in the larger system being developed. # of COTS packages * average assessment effort TBS Tailoring Activities that would have to be performed to prepare a particular COTS program for use, regardless of the system into which it is being incorporated, or even if operating as a stand-alone item. These are things such as initializing parameter values, specifying I/O screens or report formats, setting up security protocols, etc TBS Glue code Development and testing of the new code external to the COTS component itself that must be written in order to plug the component into the larger system. This code by nature is unique to the particular context in which the COTS component is being used, and must not be confused with tailoring activity as defined above. Use size of glue code in development effort models. Volatility The frequency with which new versions or updates of the COTS software being used in a larger system are released by the vendors over the course of the system's development and subsequent deployment. TBS 1.1.1. COTS Purchase Costs The cost of purchased COTS software, licenses and other expenditures are separate items to add

into the total software cost. Some of the tools include provisions for purchase costs.

Organizations often use Open Source and COTS components to reduce the cost of the software development. When they embrace this strategy, the size of the effort decreases because the number of equivalent KSLOC decreases. But, such a decrease is deceptive. Other costs creep into the equation. The most noticeable of these costs is licenses. License costs can be substantial especially if the agreement signed with the vendor calls for run-time or by seat options. In addition, labor costs associated with servicing the licenses often falls outside of the software development charter of accounts.

Example: COTS Estimation

ERP

Enterprise Resources Planning (SRDR for ERP)

Modern Estimation Challenges

Several future trends will present significant future challenges for the sizing and cost estimation of 21st century software systems. Prominent among these trends are: Rapid change, emergent requirements, and evolutionary development; Net-centric systems of systems; Model-Driven and non-developmental item (NDI)-intensive systems Ultrahigh software system assurance; Legacy maintenance and brownfield development; and Agile and kanban development. This chapter summarizes each trend and elaborates on its challenges for software sizing and cost estimation.

Rapid Change, Emergent Requirements, and Evolutionary Development

21st century software systems will encounter increasingly rapid change in their objectives, constraints, and priorities. This change will be necessary due to increasingly rapid changes in their competitive threats, technology, organizations, leadership priorities, and environments. It is thus increasingly infeasible to provide precise size and cost estimates if the systems' requirements are emergent rather than prespecifiable. This has led to increasing use of strategies such as incremental and evolutionary development, and to experiences with associated new sizing and costing phenomena such as the Incremental Development Productivity Decline. It also implies that measuring the system's size by counting the number of source lines of code (SLOC) in the delivered system may be an underestimate, as a good deal of software may be developed and deleted before delivery due to changing priorities.

There are three primary options for handling these sizing and estimation challenges. The first is to improve the ability to estimate requirements volatility during development via improved data collection and analysis, such as the use of code counters able to count numbers of SLOC added, modified, and deleted during development [Nguyen 2010]. If such data is unavailable, the best one can do is to estimate ranges of requirements volatility. For uniformity, Table 3.1 presents a recommended set of requirements volatility (RVOL) ranges over the development period for rating levels of 1 (Very Low) to 5 (Very High), such as in the DoD SRDR form [DCARC 2005].

Rating Level RVOL Range RVOL Average Very Low 0-6% 3% Low 6-12% 9% Nominal 12-24% 18% High 24-48% 36% Very High >48% 72%

Table 3.1 Recommended Requirements Volatility (RVOL) Rating Level Ranges

For incremental and evolutionary development projects, the second option is to treat the earlier increments as reused software, and to apply reuse factors to them (such as the percent of the design, code, and integration modified, perhaps adjusted for degree of software understandability and programmer unfamiliarity [Boehm et al. 2000]). This can be done either uniformly across the set of previous increments, of by having these factors vary by previous increment or by subsystem. This will produce an equivalent-SLOC (ESLOC) size for the effect of modifying the previous increments, to be added to the size of the new increment in estimating effort for the new increment. In tracking the size of the overall system, it is important to remember that these ESLOC are not actual lines of code to be included in the size of the next release.

The third option is to include an Incremental Development Productivity Decline (IDPD) factor, or perhaps multiple factors varying by increment or subsystem. Unlike hardware, where unit costs tend to decrease with added production volume, the unit costs of later software increments tend to increase, due to previous-increment breakage and usage feedback, and due to increased integration and test effort. Thus, using hardware-driven or traditional software-driven estimation methods for later increments will lead to underestimates and overruns in both cost and schedule. A relevant example was a large defense software system that had the following characteristics: • 5 builds, 7 years, $100M • Build 1 producibility over 300 SLOC/person-month • Build 5 producibility under 150 SLOC/person-month o Including Build 1-4 breakage, integration, rework o 318% change in requirements across all builds

A factor-of-2 decrease in producibility across four new builds corresponds to an average build-to-build IDPD factor of 19%. A recent quantitative IDPD analysis of a smaller software system yielded an IDPD of 14%, with significant variations from increment to increment [Tan et al. 2009]. Similar IDPD phenomena have been found for large commercial software such as the multi-year slippages in the

delivery of Microsoft's Word for Windows [Gill-Iansiti 1994] and Windows Vista, and for large agile-development projects that assumed a zero IDPD factor [Elssamadisy-Schalliol 2002]. Based on experience with similar projects, the following impact causes and ranges per increment are conservatively stated:

Less effort due to more experienced personnel, assuming reasonable initial experience level Variation depending on personnel turnover rates 5-20% More effort due to code base growth Breakage, maintenance of full code base 20-40% Diseconomies of scale in development, integration 10-25% Requirements volatility, user requests 10-25%

Table 3.2. IDPD Effort Drivers

In the best case, there would be 20% more effort (from above -20+20+10+10); for a 4-build system, the IDPD would be 6%. In the worst case, there would be 85% more effort (from above 40+25+25-5); for a 4-build system, the IDPD would be 23%. In any case, with fixed staff size, there would be either a schedule increase or incomplete builds. The difference between 6% and 23% may not look too serious, but the cumulative effects on schedule across a number of builds is very serious.

A simplified illustrative model relating productivity decline to number of builds needed to reach 4M ESLOC across 4 builds follows. Assume that the two year Build 1 production of 1M SLOC can be developed at 200 SLOC/PM. This means it will need 208 developers (500 PM/ 24 mo.). Assuming a constant staff size of 208 for all builds. the analysis shown in Figure 3.2 shows the impact on the amount of software delivered per build and the resulting effect on the overall delivery schedule as a function of the IDPD factor. Many incremental development cost estimates assume an IDPD of zero, and an on-time delivery of 4M SLOC in 4 builds. However, as the IDPD factor increases and the staffing level remains constant, the productivity decline per build stretches the schedule out to twice as long for an IDPD of 20%.

Thus, it is important to understand the IDPD factor and its influence when doing incremental or evolutionary development. Ongoing research indicates that the magnitude of the IDPD factor may vary by type of application (infrastructure software having higher IDPDs since it tends to be tightly coupled and touches everything; applications software having lower IDPDs if it is architected to be loosely coupled), or by recency of the build (older builds may be more stable). Further data collection and analysis would be very helpful in improving the understanding of the IDPD factor.

Figure 3.2. Effects of IDPD on Number of Builds to achieve 4M SLOC

Net-centric Systems of Systems (NCSoS)

If one is developing software components for use in a NCSoS, changes in the interfaces between the component systems and independently-evolving NCSoS-internal or NCSoS-external systems will add further effort. The amount of effort may vary by the tightness of the coupling among the systems; the complexity, dynamism, and compatibility of purpose of the independently-evolving systems; and the degree of control that the NCSoS protagonist has over the various component systems. The latter ranges from Directed SoS (strong control), through Acknowledged (partial control) and Collaborative (shared interests) SoSs , to Virtual SoSs (no guarantees) [USD(AT&L) 2008].

For estimation, one option is to use requirements volatility as a way to assess increased effort. Another is to use existing models such as COSYSMO [Valerdi 2008] to estimate the added coordination effort across the NCSoS [Lane 2009]. A third approach is to have separate models for estimating the systems engineering, NCSoS component systems development, and NCSoS component systems integration to estimate the added effort [Lane-Boehm 2007].

Model-Driven and Non-Developmental Item (NDI)-Intensive Development

Model-driven development and Non-Developmental Item (NDI)-intensive development are two approaches that enable large portions of software-intensive systems to be generated from model directives or provided by NDIs such as commercial-off-the-shelf (COTS) components, open source components, and purchased services such as Cloud services. Figure 3.2 shows recent trends in the growth of COTS-based applications (CBAs) [Yang et al. 2005] and services-intensive systems [Koolmanojwong-Boehm 2010] in the area of web-based e-services.

Figure 3.2. COTS and Services-Intensive Systems Growth in USC E-Services Projects

Such applications are highly cost-effective, but present several sizing and cost estimation challenges: • Model directives generate source code in Java, C++, or other third-generation languages, but unless the generated SLOC are going to be used for system maintenance, their size as counted by code counters should not be used for development or maintenance cost estimation. • Counting model directives is possible for some types of model-driven development, but presents significant challenges for others (e.g., GUI builders). • Except for customer-furnished or open-source software that is expected to be modified, the size of NDI components should not be used for estimating. • A significant challenge is to find appropriately effective size measures for such NDI components. One approach is to use the number and complexity of their interfaces with each other or with the software being developed. Another is to count the amount of glue-code SLOC being developed to integrate the NDI components, with the proviso that such glue code tends to

be about 3 times as expensive per SLOC as regularly-developed code [Basili-Boehm, 2001]. A similar approach is to use the interface elements of function points for sizing [Galorath-Evans 2006]. • A further challenge is that much of the effort in using NDI is expended in assessing candidate NDI components and in tailoring them to the given application. Some initial guidelines for estimating such effort are provided in the COCOTS model [Abts 2004]. • Another challenge is that the effects of COTS and Cloud-services evolution are generally underestimated during software maintenance. COTS products generally provide significant new releases on the average of about every 10 months, and generally become unsupported after three new releases. With Cloud services, one does not have the option to decline new releases, and updates occur more frequently. One way to estimate this source of effort is to consider it as a form of requirements volatility. • Another serious concern is that functional size measures such as function points, use cases, or requirements will be highly unreliable until it is known how much of the functionality is going to be provided by NDI components or Cloud services.

Ultrahigh Software Systems Assurance

The increasing criticality of software to the safety of transportation vehicles, medical equipment, or financial resources; the security of private or confidential information; and the assurance of “24/7” Internet, web, or Cloud services will require further investments in the development and certification of software than are provided by most current software-intensive systems. While it is widely held that ultrahigh-assurance software will substantially raise software–project cost, different models vary in estimating the added cost. For example, [Bisignani-Reed 1988] estimates that engineering highly–secure software will increase costs by a factor of 8; the 1990’s Softcost-R model estimates a factor of 3.43 [Reifer 2002]; the SEER model uses a similar value of 3.47 [Galorath-Evans 2006].

A recent experimental extension of the COCOMO II model called COSECMO used the 7 Evaluated Assurance Levels (EALs) in the ISO Standard Common Criteria for Information Technology Security Evaluation (CC) [ISO 1999], and quoted prices for certifying various EAL security levels to provide an initial estimation model in this context [Colbert-Boehm 2008]. Its added-effort estimates were a function of both EAL level and software size: its multipliers for a 5000-SLOC secure system were 1.50 for EAL 4 and 8.8 for EAL 7.

A further sizing challenge for ultrahigh-assurance software is that it requires more functionality for such functions as security audit, communication, cryptographic support, data protection, etc. These may be furnished by NDI components or may need to be developed for special systems.

Legacy Maintenance and Brownfield Development

Fewer and fewer software-intensive systems have the luxury of starting with a clean sheet of paper or whiteboard on which to create a new Greenfield system. Most software-intensive systems are already in maintenance; [Booch 2009] estimates that there are roughly 200 billion SLOC in service worldwide. Also, most new applications need to consider continuity of service from the legacy system(s) they are replacing. Many such applications involving incremental development have failed because there was no way to separate out the incremental legacy system capabilities that were being replaced. Thus, such applications need to use a Brownfield development approach that concurrently architect the new version and its increments, while re-engineering the legacy software to accommodate the incremental phase-in of the new capabilities [Hopkins-Jenkins 2008; Lewis et al. 2008; Boehm 2009].

Traditional software maintenance sizing models have determined an equivalent SLOC size by multiplying the size of the legacy system by its Annual Change Traffic ACT) fraction (% of SLOC added + % of SLOC modified)/100. The resulting equivalent size is used to determine a nominal cost of a year of maintenance, which is then adjusted by maintenance-oriented effort multipliers. These are generally similar or the same as those for development, except for some, such as required reliability and degree of documentation, in which larger development investments will yield relative maintenance savings. Some models such as SEER [Galorath-Evans 2006] include further maintenance parameters such as personnel and environment differences. An excellent summaru of software maintenance estimation is in [Stutzke 2005].

However, as legacy systems become larger and larger (a full-up BMW contains roughly 100 million SLOC [Broy 2010]), the ACT approach becomes less stable. The difference between an ACT of 1% and an ACT of 2% when applied to 100 million SLOC is 1 million SLOC. A recent revision of the COCOMO II software maintenance model sizes a new release as ESLOC = 2*(Modified SLOC) + Added SLOC + 0.5* (Deleted SLOC). The coefficients are rounded values determined from the analysis of data from 24 maintenance activities [Nguyen, 2010], in which the modified, added, and deleted SLOC were obtained from a code counting tool. This model can also be used to estimate the equivalent size of re-engineering legacy software in Brownfield software development. At first, the estimates of legacy SLOC modified, added, and deleted will be very rough, and can be refined as the design of the maintenance modifications or Brownfield re-engineering is determined.

Agile and Kanban Development

The difficulties of software maintenance estimation can often be mitigated by using workflow management techniques such as Kanban [Anderson 2010]. In Kanban, individual maintenance upgrades are given Kanban cards (Kanban is the Japanese word for card; the approach originated with the Toyota Production System). Workflow management is accomplished by limiting the number of cards introduced into the development process, and pulling the cards into the next stage of development (design, code, test, release) when open capacity is available (each stage has a limit of the number of cards it can be processing at a given time). Any buildups of upgrade queues waiting to be pulled forward are given management attention to find and fix bottleneck root causes or to rebalance the

manpower devoted to each stage of development. A key Kanban principle is to minimize work in progress.

An advantage of Kanban is that if upgrade requests are relatively small and uniform, that there is no need to estimate their required effort; they are pulled through the stages as capacity is available, and if the capacities of the stages are well-tuned to the traffic, work gets done on schedule. However, if a too-large upgrade is introduced into the system, it is likely to introduce delays as it progresses through the stages. Thus, some form of estimation is necessary to determine right-size upgrade units, but it does not have to be precise as long as the workflow management pulls the upgrade through the stages. For familiar systems, performers will be able to right-size the units. For Kanban in less-familiar systems, and for sizing builds in agile methods such as Scrum, group consensus techniques such as Planning Poker [Cohn 2005] or Wideband Delphi [Boehm 1981]can generally serve this purpose.

The key point here is to recognize that estimation of knowledge work can never be perfect, and to create development approaches that compensate for variations in estimation accuracy. Kanban is one such; another is the agile methods' approach of timeboxing or schedule-as-independent-variable (SAIV), in which maintenance upgrades or incremental development features are prioritized, and the increment architected to enable dropping of features to meet a fixed delivery date (With Kanban, prioritization occurs in determining which of a backlog of desired upgrade features gets the next card). Such prioritization is a form of value-based software engineering, in that the higher-priority features can be flowed more rapidly through Kanban stages [Anderson 2010], or in general given more attention in defect detection and removal via value-based inspections or testing [Boehm-Lee 2005; Li-Boehm 2010]. Another important point is that the ability to compensate for rough estimates does not mean that data on project performance does not need to be collected and analyzed. It is even more important as a sound source of continuous improvement and change adaptability efforts.

Putting It All Together at the Large-Project or Enterprise Level

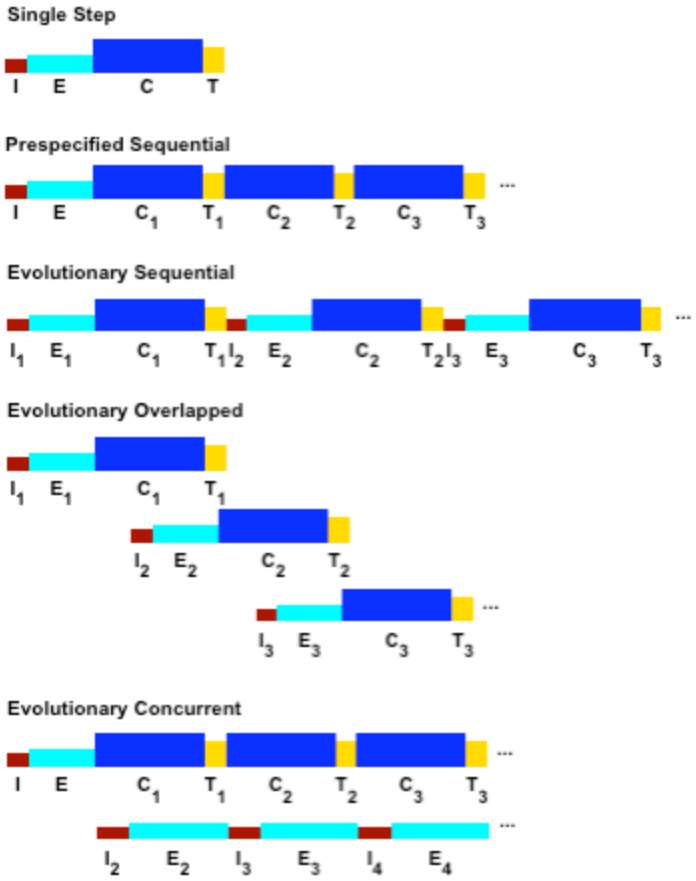
The biggest challenge of all is that the six challenges above need to be addressed concurrently. Suboptimizing on individual-project agility runs the risks of easiest-first lock-in to unscalable or unsecurable systems, or of producing numerous incompatible stovepipe applications. Suboptimizing on security assurance and certification runs the risks of missing early-adopter market windows, of rapidly responding to competitive threats, or of creating inflexible, user-unfriendly systems.

One key strategy for addressing such estimation and performance challenges is to recognize that large systems and enterprises are composed of subsystems that have different need priorities and can be handled by different estimation and performance approaches. Real-time, safety-critical control systems and security kernels need high assurance, but are relatively stable. GUIs need rapid adaptability to change, but with GUI-builder systems, can largely compensate for lower assurance levels via rapid fixes. A key point here is that for most enterprises and large systems, there is no one-size-fits-all method of sizing, estimating, and performing.

5.7.1. Estimation Approaches for Different Processes

This implies a need for guidance on what kind of process to use for what kind of system or subsystem, and on what kinds of sizing and estimation capab

The figure below summarizes the traditional single-step waterfall process plus several forms of incremental development, each of which meets different competitive challenges and which are best served by different cost estimation approaches. The time phasing of each form is expressed in terms of the increment 1, 2, 3, … content with respect to the Rational Unified Process (RUP) phases of Inception, Elaboration, Construction, and Transition (IECT):



The Single Step model is the traditional waterfall model, in which the requirements are prespecified, and the system is developed to the requirements in a single increment. Single-increment parametric estimation models, complemented by expert judgment, are best for this process.

The Prespecified Sequential incremental development model is not evolutionary. It just splits up the development in order to field an early Initial Operational Capability, followed by several pre-planned product Improvements (P3Is). When requirements are prespecifiable and stable, it enables a strong, predictable process. When requirements are emergent and/or rapidly changing, it often requires very expensive rework when it needs to undo architectural commitments. Cost estimation can be performed by sequential application of single-step parametric models plus the use of an IDPD factor, or by parametric model extensions supporting the estimation of increments, including options for increment overlap and breakage of existing increments, such as the extension of COCOMO II Incremental Development Model (COINCOMO) extension described in Appendix B of [Boehm et al. 2000].

The Evolutionary Sequential model rapidly develops an initial operational capability and upgrades it based on operational experience. Pure agile software development fits this model: if something is wrong, it will be fixed in 30 days in the next release. Rapid fielding also fits this model for larger or hardware-software systems. Its strength is getting quick-response capabilities in the field. For pure agile, it can fall prey to an easiest-first set of architectural commitments which break when, for example, it tries to add security or scalability as a new feature in a later increment. For rapid fielding, it may be expensive to keep the development team together while waiting for usage feedback, but it may be worth it. For small agile projects, group consensus techniques such as Planning Poker are best; for larger projects, parametric models with an IDPD factor are best.

Evolutionary Overlapped covers the special case of deferring the next increment until critical enablers such as desired new technology, anticipated new commercial product capabilities, or needed funding become available or mature enough to be added.

Evolutionary Concurrent has the systems engineers handling the change traffic and rebaselining the plans and specifications for the next increment, while keeping the development stabilized for the current increment. Its example and pros and cons are provided in chart 10.

Type Examples Pros Cons Cost Estimation Single Step Stable; High Assurance Prespecifiable full-capability requirements Emergent requirements or rapid change Single-increment parametric estimation models Prespecified Sequential Platform base plus PPPIs Prespecifiable full-capability requirements Emergent requirements or rapid change COINCOMO or repeated single-increment parametric model estimation with IDPD Evolutionary Sequential Small: Agile Large: Evolutionary Development Adaptability to change Easiest-first; late, costly breakage Small: Planning-poker-type Large: Parametric with IDPD and Requirements Volatility Evolutionary Overlapped COTS-intensive systems Immaturity risk avoidance Delay may be noncompetitive Parametric with IDPD and Requirements Volatility Evolutionary Concurrent Mainstream product lines; Systems of systems High assurance with rapid change Highly coupled systems with very rapid change COINCOMO with IDPD for development; COSYSMO for rebaselining Table 3.3 Situation-Dependent

Processes and Estimation Approaches All Cost Estimation approaches also include an expert-judgment cross-check

Table 3.4 provides criteria for deciding which of the four classes of incremental and evolutionary acquisition (EvA) defined in Table 3.3 to use, plus the choice of non-incremental, single-step development.

The Single-Step-to-Full-Capability process exemplified by the traditional waterfall or sequential Vee model is appropriate if the product's requirements are prespecifiable and have a low probability of significant change; and if there is no value in or opportunity to deliver a partial product capability. A good example would be the hardware portion of a geosynchronous satellite.

The Prespecified Sequential process is best if the product's requirements are prespecifiable and have a low probability of significant change; and if waiting for the full system to be developed incurs a loss of important and deliverable incremental mission capabilities. A good example would be a well-understood and well-prioritized sequence of software upgrades to a programmable radio.

The Evolutionary Sequential process is best when there is a need to get operational feedback on a quick-response capability before defining and developing the next increment's content. Agile methods fit into this category, as do systems undergoing rapid competitive change.

The Evolutionary Overlapped process is best when one does not need to wait for operational feedback, but may need to wait for next-increment enablers such as technology maturity, external system capabilities, or needed resources. A good example is the need to wait for a mature release of an anticipated commercial product. The Evolutionary Concurrent process is best when the enablers are available, but there is a great deal of change traffic to be handled that would destabilize the team developing the current increment. Examples may be new competitive threats, emergent user capability needs, external system interface changes, technology matured on other programs, or COTS upgrades. Table 3.4. Process Model Decision Table

- Example enablers: Technology maturity; External-system capabilities; Needed resources

References

Appendices

Acronyms

AAF
: Adaptation Adjustment Factor is used with adapted software to produce an equivalent size. It includes the effects of Design Modified (DM), Code Modified (CM), and Integration Modifified (IM).

AAM
: Adaptation Adjustment Multiplier

CER
: Cost Estimating Relationship

CM
: Code Modified

COCOMO
: Constructive Cost Model

DM
: Design Modified

IM
: Integration Modified

SEER
: A tool suite produced by Galorath

SLIM
: A tool suite produced by Quantitative Software Management

SU
: Software Understanding

UNFM
: Programmer Unfamiliarity

UCC
: Universal Code Counter

USC
: University of Southern California

4GL
: Fourth Generation Language

ACEIT
: Automated Cost Estimating Integrated Tools

ACWP
: Actual Cost of Work Performed

AMS
: Acquisition Management System

BCWP
: Budgeted Cost of Work Performed

BCWS
: Budgeted Cost of Work Scheduled

BFP
: Basic Feature Point

CDRL
: Contract Data Requirements List

CDR
: Critical Design Review

CER
: Cost Estimating Relationship

CMM
: Capability Maturity Model

CO
: Contracting Officer

COCOMO
: COnstruction COst Model

COCOTS
: COnstruction COTS

COTS
: Commercial-off-the Shelf

CPM
: Critical Path Method

C/SCSC
: Costs/Schedule Control System Criteria

CSC
: Computer Software Component

CSCI
: Computer Software Configuration Item

CSU
: Computer Software Unit

DDE
: Dynamic Data Exchange

DoD
: Department of Defense

EA
: Evolutionary Acquisition

EI
: External Inputs

EIF
: External Interfaces

EO
: External Outputs

EQ
: External Inquiries

EVMS
: Earned Value Management System

FAA CEH
: FAA Cost Estimating Handbook

FAA PH
: FAA Pricing Handbook

FAQ
: Frequently Asked Questions

FCA
: Functional Configuration Audit

FPA
: Function Point Analysis

FPC
: Function Point Count

FPH
: FAA Pricing Handbook

GAO
: U.S. General Accounting Office

GUI
: Graphical User Interface

HWCI
: Hardware Configuration item

HOL
: Higher Order Language

ICE
: Independent Cost Estimate

IEEE
: Institute of Electrical and Electronics Engineers

IFPUG
: International Function Point User's Group

ILF
: Internal Files

IRS
: Interface Requirement Specification

IS
: Information System

KDSI: Thousands Delivered Source Instructions

LCC
: Life Cycle Cost

MTTD
: Mean-Time-To-Detect

NASA
: National Aeronautics and Space Administration

NCCA
: Naval Center for Cost Analysis

NRaD
: United States Navy's Naval Command, Control, Surveillance Center, RDT&E Division, Software Engineering Process Office

OO
: Object Oriented

PCA
: Physical Configuration Audit

PERT
: Program Evaluation and Review Technique

SDD
: Software Design Document

SDP
: Software Development Plan

SDR
: Software Design Review

SEER-SEM
: System Evaluation and Estimation of Resources Software Estimating Model

SEI
: Software Engineering Institute

SER
: Schedule Estimating Relationship

SLIM
: Software Life Cycle Model

SLOC
: Source Lines of Code

SRR
: Systems Requirements Review

SRS
: Software Requirements Specification

SSCAG
: Space Systems Cost Analysis Group

SSR
: Software Specification Review

SSS
: System Segment Specification

WBS
: Work Breakdown Structure

Cost Factors

Cost Model Comparisons for SLOC Sizing

Size Units

The primary unit of software size in the effort models is Thousands of Source Lines of Code (KSLOC). KSLOC can be converted from other size measures, and additional size units can be used directly in the models as described next. User-defined proxy sizes can be developed for any of the models.

COCOMO II

The COCOMO II size model is based on SLOC or function points converted to SLOC, and can be calibrated and used with other software size units. Examples include use cases, use case points, object points, physical lines, and others. Alternative size measures can be converted to lines of code and used directly in the model or it can be independently calibrated directly to different measures.

SEER-SEM

Several sizing units can be used alone or in combination. SEER can use SLOC, function points and custom proxies. COTS elements are sized with *Features* and *Quick Size*. SEER allows proxies as a flexible way to estimate software size. Any countable artifact can be established as measure. Custom proxies can be used with other size measures in a project. Available pre-defined proxies that come with SEER include *Web Site Development*, *Mark II Function Point, Function Points* (for direct IFPUG-standard function points) and *Object-Oriented Sizing*.

SEER converts all size data into internal size units, also called effort units, Sizing in SEER-SEM can be based on function points, source lines of code, or user-defined metrics. Users can combine or select a single metric for any project element or for the entire project. COTS WBS elements also have specific size inputs defined either by Features, Object Sizing, or Quick Size, which describe the functionality being integrated.

New Lines of Code are the original lines created for the first time from scratch.

Pre-Existing software is that which is modified to fit into a new system. There are two categories of pre-existing software:

- *Pre-existing, Designed for Reuse*
- *Pre-existing, Not Designed for Reuse.*

Both categories of pre-existing code then have the following subcategories:

Pre-existing lines of code which is the number of lines from a previous system

Lines to be Deleted are those lines deleted from a previous system.

Redesign Required is the percentage of existing code that must be redesigned to meet new system requirements.

Reimplementation Required is the percentage of existing code that must be re-implemented, physically recoded, or reentered into the system, such as code that will be translated into another language.

Retest Required is the percentage of existing code that must be retested to ensure that it is functioning properly in the new system.

SEER then uses different proportional weights with these parameters in their AAF equation according to

*Pre-existing Effective Size = (0.4*A + 0.25*B + 0.35*C).*

Where A, B and C are the respective percentages of code redesign, code reimplementation, and code retest required.

Function-Based Sizing

- External Input (EI)
- External Output (EO)
- Internal Logical File (ILF)
- External Interface Files (EIF)
- External Inquiry (EQ)

- Internal Functions (IF) ‚Äì any functions that are neither data nor transactions

Proxies

- Web Site Development
- Mark II Function Points
- Function Points (direct)
- Object-Oriented Sizing.

COTS Elements

- Quick Size
- Application Type Parameter
- Functionality Required Parameter
- Features
- Number of Features Used
- Unique Functions
- Data Tables Referenced
- Data Tables Configured

True S

The True S software cost model size measures may be expressed in different size units including source lines of code (SLOC), function points, predictive object points (POPs) or Use Case Conversion Points (UCCPs). True S also differentiates executable from non-executable software sizes. *Functional Size* describes software size in terms of the functional requirements that you expect a Software COTS component to satisfy. Also see section TBD. The True S software cost model size definitions for all of the size units are listed below.

Adapted Code Size - This describes the amount of existing code that must be changed, deleted, or adapted for use in the new software project. When the value is zero (0.00), the value for New Code Size or Reused Code Size must be greater than zero.

Adapted Size Non-executable - This value represents the percentage of the adapted code size that is non-executable (such as data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. When a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Amount for Modification - This represents the percent of the component functionality that you plan to modify, if any. The Amount for Modification value (like Glue Code Size) affects the effort calculated for the Software Design, Code and Unit Test, Perform Software Integration and Test, and Perform Software Qualification Test activities.

Auto Gen Size Non-executable - This value represents the percentage of the Auto Generated Code Size that is non-executable (such as, data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Auto Generated Code Size - This value describes the amount of code generated by an automated design tool for inclusion in this component.

Auto Trans Size Non-executable - This value represents the percentage of the Auto Translated Code Size that is non-executable (such as, data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Auto Translated Code Size - This value describes the amount of code translated from one programming language to another by using an automated translation tool (for inclusion in this component).

Auto Translation Tool Efficiency - This value represents the percentage of code translation that is actually accomplished by the tool. More efficient auto translation tools require more time to configure the tool to translate. Less efficient tools require more time for code and unit test on code that is not translated.

Code Removal Complexity - This value describes the difficulty of deleting code from the adapted code. Two things need to be considered when deleting code from an application or component: the amount of functionality being removed and how tightly or loosely this functionality is coupled with the rest of the system. Even if a large amount of functionality is being removed, if it accessed through a

single point rather than from many points, the complexity of the integration will be reduced.

Deleted Code Size - This describes the amount of pre-existing code that you plan to remove from the adapted code during the software project. The Deleted Code Size value represents code that is included in Adapted Code Size, therefore, it must be less than, or equal to, the Adapted Code Size value.

Equivalent Source Lines of Code - The ESLOC (Equivalent Source Lines of Code) value describes the magnitude of a selected cost object in Equivalent Source Lines of Code size units. True S does not use ESLOC in routine model calculations, but provides an ESLOC value for any selected cost object. Different organizations use different formulas to calculate ESLOC.

The True S calculation for ESLOC is

$$ESLOC = NewCode + .70(AdaptedCode) + .10(ReusedCode).$$

To calculate ESLOC for a Software COTS, True S first converts Functional Size and Glue Code Size inputs to SLOC using a default set of conversion rates. NewCode includes Glue Code Size and Functional Size when the value of Amount for Modification is greater than or equal to 25%. AdaptedCode includes Functional Size when the value of Amount for Modification is less than 25% and greater than zero. ReusedCode includes Functional Size when the value of Amount for Modification equals zero.

Functional Size - This value describes software size in terms of the functional requirements that you expect a Software COTS component to satisfy. When you select Functional Size as the unit of measure (Size Units value) to describe a Software COTS component, the Functional Size value represents a conceptual level size that is based on the functional categories of the software (such as Mathematical, Data Processing, or Operating System). A measure of Functional Size can also be specified using Source Lines of Code, Function Points, Predictive Object Points or Use Case Conversion Points if one of these is the Size Unit selected.

Glue Code Size - This value represents the amount of glue code that will be written. Glue Code holds the system together, provides interfaces between Software COTS components, interprets return codes, and translates data into the proper format. Also, Glue Code may be required to compensate for inadequacies or errors in the COTS component selected to deliver desired functionality.

New Code Size - This value describes the amount of entirely new code that does not reuse any design, code, or test artifacts. When the value is zero (0.00), the value must be greater than zero for Reused Code Size or Adapted Code Size.

New Size Non-executable - This value describes the percentage of the New Code Size that is non-executable (such as data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.0 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

Percent of Code Adapted - This represents the percentage of the adapted code that must change to enable the adapted code to function and meet the software project requirements.

Percent of Design Adapted - This represents the percentage of the existing (adapted code) design that must change to enable the adapted code to function and meet the software project requirements. This value describes the planned redesign of adapted code. Redesign includes architectural design changes, detailed design changes, and any necessary reverse engineering.

Percent of Test Adapted - This represents the percentage of the adapted code test artifacts that must change. Test plans and other artifacts must change to ensure that software that contains adapted code meets the performance specifications of the Software Component cost object.

Reused Code Size - This value describes the amount of pre-existing, functional code that requires no design or implementation changes to function in the new software project. When the value is zero (0.00), the value must be greater than zero for New Code Size or Adapted Code Size.

Reused Size Non-executable - This value represents the percentage of the Reused Code Size that is non-executable (such as, data statements, type declarations, and other non-procedural statements). Typical values for fourth generation languages range from 5.00 percent to 30.00 percent. If a value cannot be obtained by any other means, the suggested nominal value for non-executable code is 15.00 percent.

SLIM

SLIM uses effective system size composed of new, modified and reused code. Deleted code is not considered in the model. If there is reused code than the Productivity Index (PI) factor may be adjusted to add in time and effort for regression testing and integration of the reused code.

SLIM provides different sizing techniques including:

- Sizing by history
- Total system mapping
- Sizing by decomposition

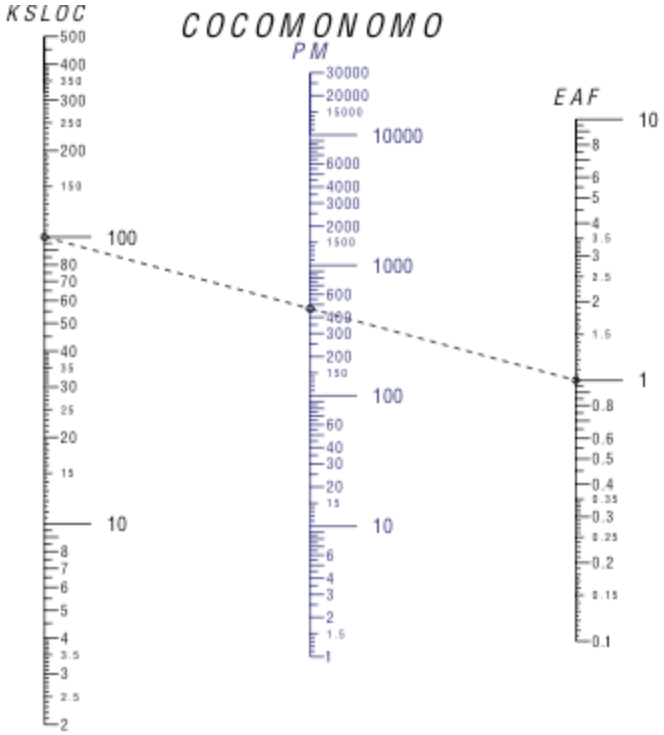
- Sizing by module
- Function point sizing.

Alternative sizes to SLOC such as use cases or requirements can be used in Total System Mapping. The user defines the method and quantitative mapping factor.

Nomograms

Following are nomograph downloads for visual computations of cost and schedule. Cost estimating relationships (CERs) are exhibited for two variable CERs and three variable CERs. The files are vector format PDFs that scale for different size paper and screens without losing resolution.

Example: Draw a line between any two variables to compute the third. This nomograph computes effort, size or the effort adjustment factor for the COCOMO model.



Cost Models

COCOMO II (2 - 500 KSLOC)

Three variables for Size (KSLOC), Effort (Person-Months), and Effort Adjustment Factor (EAF).

Domain Analysis

Communications

SRDR Communications - Size and Effort

MIL-STD-881C WBS Mapping to Productivity Types

The Work Breakdown Structures were adapted from MIL-STD-881C to assist in determining the correct Productivity Type (PT). Each

System from 881C is listed with the associated one of more Metrics Manual Operating Environments.

Within the environments, look through the Subsystems to find one that matches the component being estimated. Each Subsystem or Sub-Subsystem has a matching PT.

Use the PT to lookup the associated Productivity-based CER and Model-based CER/SER.

Aircraft Systems (881C Appendix-A)

MAV: Manned Aerial Vehicle

Env	SubSys	Sub-Subsystem	Domain	PT
MAV	Air Vehicle	Flight Control Subsystem	⇐	VC
MAV		Auxiliary Power Subsystem	⇐	VC
MAV		Hydraulic Subsystem	⇐	VC
MAV		Electrical Subsystem	⇐	VC
MAV		Crew Station Subsystem	⇐	VC
MAV		Environmental Control Subsystem	⇐	VC
MAV		Fuel Subsystem	⇐	VC
MAV		Landing Gear	⇐	VC
MAV		Rotor Group	⇐	VC
MAV		Drive System	⇐	VC
MAV	Avionics	Communication/Identification	Intercoms	RTE
MAV			Radio System(S)	RTE
MAV			Identification Equipment (IFF)	RTE
MAV			Data Links	RTE
MAV		Navigation/Guidance	Radar	SCP
MAV			Radio	SCP
MAV			Other Essential Nav Equipment	RTE
MAV			Radar Altimeter	SCP
MAV			Direction Finding Set	RTE
MAV			Doppler Compass	SCP
MAV		Mission Computer/Processing	⇐	MP
MAV		Fire Control	Search, Target, Tracking Sensors	SSP
MAV			Self-Contained Navigation	RTE
MAV			Self-Contained Air Data Systems	RTE
MAV			Displays, Scopes, Or Sights	RTE
MAV			Bombing Computer	MP
MAV			safety Devices	RTE
MAV		Data Display and Controls	Multi-Function Displays	RTE
MAV			Control Display Units	RTE
MAV			Display Processors	MP
MAV			On-Board Mission Planning	TRN
MAV		Survivability	Ferret And Search Receivers	SCP
MAV			Warning Devices	SCP
MAV			Electronic Countermeasures	SCP
MAV			Jamming Transmitters	SCP

MAV			Chaff	SCP
MAV			Infra-Red Jammers	SCP
MAV			Terrain-Following Radar	SCP
MAV		Reconnaissance	Photographic Sensors	SCP
MAV			Electronic Sensors	SCP
MAV			Infrared Sensors	SCP
MAV			Search Receivers	SCP
MAV			Recorders	SCP
MAV			Warning Devices	SCP
MAV			Magazines	RTE
MAV			Data Link	RTE
MAV		Automatic Flight Control	Flight Control Computers	MP
MAV			Signal Processors	SCP
MAV			Data Formatting	MP
MAV			Interfaces To Other Systems	MP
MAV			Pressure Transducers	SCP
MAV			Rate Gyros	SCP
MAV			Accelerometers	SCP
MAV			Motion Sensors	SCP
MAV		Health Monitoring System	⇐	SYS
MAV		Stores Management	⇐	MP

Missile Systems (881C Appendix-C)

UOV: Unmanned Ordinance Vehicle

Missile Systems (881C Appendix-C) UOV: Unmanned Ordinance Vehicle

Env	SubSystem	Sub-Subsystem	Domain	PT
UOV	Air Vehicle	Guidance	Seeker Assembly	SCP
UOV			Guidance Software	RTE
UOV		Navigation	Sensor Assembly	SCP
UOV			Navigation Software	RTE
UOV		Payload	Target Defeat Mechanism	RTE
UOV			Target Detection Device	SCP
UOV			Fuze	SCP
UOV			Payload-specific software	VP
UOV		Power and Distribution	Primary Power	VC
UOV			Power Conditioning Electronics	VC
UOV			Power and distribution software	VC
UOV		Communications	Antenna Assembly	SCP
UOV			Communications software	RTE
UOV		Propulsion Subsystem	Motor Engine	VC
UOV			Thrust Vector Actuation	VC

UOV			Attitute Control System	VC
UOV			Fuel/Oxidizer Liquid Management	VC
UOV			Arm/Fire Device	VC
UOV			Flight Termination/Mission Termination	RTE
UOV			Propulsion software	VC
UOV		Controls	Controls software	VC
UOV		Reentry System	⇐	VC
UOV		Post boost System	⇐	VC
UOV		On Board Test Equipment	⇐	TST
UOV		On Board Training Equipment	⇐	TRN
UOV		Auxiliary Equipment	⇐	SYS
UOV		Air Vehicle Software	⇐	MP
UOV	Encasement Device	Encasement Device Software	⇐	MP
M/UAV	Command & Launch	Surveillance, Indentification, and Tracking Sensors	⇐	SCP
M/UAV		Launch & Guidance Control	⇐	RTE
M/UAV		Communications	⇐	RTE
M/UAV		Launcher Equipment	⇐	RTE
M/UAV		Auxiliary Equipment	⇐	SYS

Ordinance Systems (881C Appendix-D)

UOV: Unmanned Ordinance Vehicle

Env	SubSystem	Sub-Subsystem	Domain	PT
UOV	Munition	Guidance	Seeker Assembly	SCP
UOV			Guidance Software	RTE
UOV		Navigation	Sensor Assembly	SCP
UOV			Navigation Software	RTE
UOV		Payload	Target Defeat Mechanism	RTE
UOV			Target Detection Device	SCP
UOV			Fuze	SCP
UOV			Payload software	VP
UOV		Power and Distribution	Primary Power	VC
UOV			Power Conditioning Electronics	VC
UOV			Power and distribution software	VC
UOV		Communications	Antenna Assembly	SCP
UOV			Communications software	RTE
UOV		Propulsion Subsystem	Motor Engine	VC
UOV			Fuel/Oxidizer Liquid Management	VC
UOV			Arm/Fire Device	VC
UOV			Thrust Vector Actuation	VC
UOV			Flight Termination/Mission Termination	RTE
UOV			Propulsion software	VC

UOV		Controls	Controls software	VC
UOV		On Board Test Equipment	⇐	TST
UOV		On Board Training Equipment	⇐	TRN
UOV		Auxiliary Equipment	⇐	SYS
UOV		Munition Software	⇐	MP
UOV	Launch System	Fire Control	⇐	RTE

Sea Systems (881C Appendix-E)

MMV: Manned Maritime Vessel

Sea Systems (881C Appendix E) - Environment: Manned Maritime Vessel (MMV)<!--EndFragment-->

Env	SubSystem	Sub-Subsystem	Domain	PT
MMV	Ship	Command, Communication & Surveillance	Sensing and data	RTE
MMV			Navigation equipment	RTE
MMV			Interior communication	RTE
MMV			Gun fire control system	RTE
MMV			Non-electronic & electronic countermeasure	RTE
MMV			Missile fire control systems	RTE
MMV			Antisubmarine warfare fire control and torpedo fire control systems	RTE
MMV			Radar systems	RTE
MMV			Radio communication systems	RTE
MMV			Electronic navigation systems	RTE
MMV			Space vehicle electronic tracking systems	RTE
MMV			Sonar systems	RTE
MMV			Electronic tactical data systems	MP
MMV			Fiber optic plant	BIS
MMV			Inter/intranet	BIS
MMV			Entertainment systems	BIS

Space Systems (881C Appendix-F)

MSV: Manned Space Vehicle

USV: Unmanned Space Vehicle

MGS: Manned Ground Site

Env	SubSystem	Sub-Subsystem	Domain	PT
M/USV	Bus	Structures & Mechanisms (SMS)	⇐	VC
M/USV		Thermal Control (TCS)	⇐	VC

M/USV		Electrical Power (EPS)	⇐	VC
M/USV		Attitude Control (ACS)	⇐	VC
M/USV		Propulsion	⇐	VC
M/USV		Telemetry, Tracking, & Command (TT&C)	⇐	RTE
M/USV		Bus Flight Software	⇐	MP
M/USV	Payload	Thermal Control	⇐	RTE
M/USV		Electrical Power	⇐	RTE
M/USV		Pointing, Command, & Control Interface	⇐	VP
M/USV		Payload Antenna	⇐	SCP
M/USV		Payload Signal Electronics	⇐	SCP
M/USV		Optical Assembly	⇐	SCP
M/USV		Sensor	⇐	SCP
M/USV		Payload Flight Software	⇐	VP
MGS	Ground Operations & Processing Center	Mission Management	⇐	BIS
MGS		Command and Control	⇐	C&C
MGS		Mission Data Processing	⇐	BIS
MGS		Mission Data Analysis	⇐	BIS
MGS		Collection Management	⇐	BIS
MGS		Infrastructure & Framework	⇐	SYS
MGS	Ground Terminal /Gateway	Ground Terminal Software	Application Specific Iintegrated Circuit	SCP
MGS			Field Programmable Gate Array	SCP

Surface Vehicle Systems (881C Appendix-G)

MGV: Manned Ground Vehicle AND

UGV: Unmanned Ground Vehicle

Env	SubSystem	Sub-Subsystem	Domain	PT
M/UGV	Primary Vehicle	System Survivability	⇐	
M/UGV		Turret Assembly	⇐	RTE
M/UGV		Suspension/Steering	⇐	SCP
M/UGV		Vehicle Electronics	Computers And Other Devices For Command And Control.	VC
M/UGV			Data Control And Distribution	MP
M/UGV			Controls And Displays	BIS
M/UGV			Power Distribution And Management	RTE
M/UGV			Health Management Systems	RTE
M/UGV		Power Package/Drive Train	Controls And Instrumentation	VC
M/UGV			Power Transmission, Final Drivers, And Power Takeoffs	VC

M/UGV			Brakes And Steering When Integral To Power Transmission	VC
M/UGV			Hybrid Electric Drive Systems	VC
M/UGV			Energy Storage Systems	VC
M/UGV		Fire Control	Radars And Other Sensors	SCP
M/UGV			Controls And Displays	RTE
M/UGV			Sights Or Scopes	RTE
M/UGV			Range Finders, Gun Drives And Stabilization Systems	RTE
M/UGV		Armament	Main Gun And Secondary Guns	VP
M/UGV			Missile Launchers	VP
M/UGV			Non-Lethal Weapons	VP
M/UGV			Other Offensive Weapon Systems	VP
M/UGV		Automatic Ammunition Handling	⇐	MP
M/UGV		Navigation and Remote Piloting	⇐	RTE
M/UGV		Communications	⇐	RTE
UGV	Remote Control System (UGV specific)	Ground Control Systems	⇐	RTE
UGV		Command and Control Subsystem	⇐	C&C
UGV		Remote Control System Software	⇐	RTE

Unmanned Air Vehicle Systems (881C Appendix-H)

UAV: Unmanned Aerial Vehicle

MGS: Manned Ground Site

Env	SubSystem	Sub-Subsystem	Domain	PT
UAV	Air Vehicle	Vehicle Subsystems	Propulsion	VC
UAV			Flight Control Subsystem	VC
UAV			Auxiliary Power Subsystem	VC
UAV			Hydraulic Subsystem	VC
UAV			Electrical Subsystem	VC
UAV			Environmental Control	VC
UAV			Subsystem Fuel Subsystem	VC
UAV			Landing Gear	VC
UAV			Rotor Group	VC
UAV			Drive System	VC
UAV		Avionics	Communication/Identification	RTE
UAV			Navigation/Guidance	RTE
UAV			Automatic Flight Control	VC
UAV			Health Monitoring System	SYS

UAV			Stores Management	VP
UAV			Mission Processing	MP
UAV			Fire Control	RTE
UAV	Payload	Survivability Payload	⇐	VP
UAV		Reconnaissance Payload	⇐	VP
UAV		Electronic Warfare Payload	⇐	VP
UAV		Armament/Weapons Delivery	⇐	VP
MGS	Ground/Host Segment	Ground Control Systems	⇐	C&C
MGS		Command and Control Subsystem	⇐	RTE
MGS		Launch and Recovery Equipment	⇐	RTE

Unmanned Maritime Vessel Systems (881C Appendix-I)

UMV: Unmanned Maritime Vessel

MMV: Manned Maritime Vessel

Env	SubSystem	Sub-Subsystem	Domain	PT
UMV	Maritime Vehicle	Energy Storage / Conversion	Energy Storage And Conversion Monitoring And Control System	VC
UMV		Electrical Power	Electric Power Monitoring And Control System	VC
UMV		Vehicle Command and Control	Mission Control	RTE
UMV			Navigation	RTE
UMV			Guidance And Control	RTE
UMV			Health Status Monitoring	SYS
UMV			Rendezvous, Homing And Docking Systems	SYS
UMV			Fire Control	RTE
UMV		Surveillance	⇐	RTE
UMV		Communications/Identification	⇐	RTE
UMV		Ship Control Systems	Hovering And Depth Control	VC
UMV			Ballast And Trim	VC
UMV			Maneuvering System	VC
UMV		Auxiliary Systems	Emergency Systems	MP
UMV			Launch And Recovery System	MP
UMV			Environmental Control System	MP
UMV			Anchoring, Mooring And Towing	MP
UMV			Miscellaneous Fluid Systems	MP
UMV	Payload	Survivability Payload	⇐	VP
UMV		Intelligence, Surveillance Reconnaissance Payload	⇐	VP
UMV		Armament/Weapons Delivery Payload	⇐	VP
UMV		Mission Payload	⇐	VP
MMV	Shipboard Segment	Shipboard UM Command and Control Subsystem	⇐	C&C

MMV		Shipboard Communication Subsystem	⇐	RTE
MMV		Shipboard Power Subsystem	⇐	VC
MMV		Launch and Recovery Equipment	⇐	RTE

Launch Vehicles (881C Appendix-J)

UOV: Unmanned Ordinance Vehicle

Env	SubSystem	Sub-Subsystem	Domain	PT
UOV	Launch Vehicle	Stage(s)	Propulsions System	VC
UOV			Reaction Control System	VC
UOV			Recovery System	VC
UOV			Environmental Control System	RTE
UOV			Stage Peculiar Avionics	RTE
UOV		Avionics	Guidance Navigation and Control	RTE
UOV			Power	VC
UOV			Data Acquisition and Telemetry	RTE
UOV			Range Tracking & Safety (Airborne)	RTE
UOV			Flight Software	VC
UOV	Flight Operations	Real-time mission control	Telemetry processing	RTE
MGS			Communications	RTE
MGS			Data reduction and analysis	BIS

Automated Information Systems (881C Appendix-K)

MGS: Manned Ground Site

Env	SubSystem	Sub-Subsystem	Domain	PT
MSG	Custom Application Software	Subsystem Software CSCI	⇐	Variable
MSG	Enterprise Service Element	Software COTS/GOTS	Component identification	BIS
MSG			Assessment and Selection	BIS
MSG			Prototyping	BIS
MSG			Glue code development	BIS
MSG			Tailoring and configuration	BIS
MSG	Enterprise Information System	Business Software COTS/GOTS	Component identification	BIS
MSG			Assessment and Selection	BIS
MSG			Prototyping	BIS
MSG			Glue code development	BIS
MSG			Tailoring and configuration	BIS

Common Elements (881C Appendix-L)

Applies to ALL environments

Env	SubSystem	Sub-Subsystem	Domain	PT
CE	System Integration Lab (SIL)	SIL Software - Sil Operations	⇐	TST
CE		SIL Software - Simulation	⇐	SCI
CE	Test and Evaluation Support	Test Software	⇐	STS
CE	Automated Test Equipment	Equipment Software	⇐	TST
CE	Training	Equipment	⇐	TRN
CE		Simulators	⇐	SCI
CE		Computer Based-Application	⇐	BIS
CE		Computer Based-Web	⇐	BIS
CE	Support Equipment	Software	⇐	BIS
CE	Test and Measurement Equipment	Equipment Software	⇐	TST
CE	Data Migration	Software Utilities	⇐	BIS

References

[Abts 2004] Abts C, "Extending the COCOMO II Software Cost Model to Estimate Effort and Schedule for Software Systems Using Commercial-off-the-Shelf (COTS) Software Components: The COCOTS Model," PhD Dissertation, Department of Industrial and Systems Engineering, University of Southern California, May 2004

[Anderson 2010] Anderson D, Kanban, Blue Hole Press, 2010

[Beck 2000] Beck, K., Extreme Programming Explained, Addison-Wesley, 2000.

===[Boehm 1981]=== Boehm B., Software Engineering Economics. Englewood Cliffs, NJ, Prentice-Hall, 1981

[Boehm et al. 2000] Boehm B., Abts C., Brown W., Chulani S., Clark B., Horowitz E., Madachy R., Reifer D., Steece B., Software Cost Estimation with COCOMO II, Prentice-Hall, 2000

[Boehm et al. 2000b] Boehm B, Abts C, Chulani S, “Software Development Cost Estimation Approaches – A Survey”, USC-CSE-00-505, 2000

[Boehm et al. 2004] Boehm B, Bhuta J, Garlan D, Gradman E, Huang L, Lam A, Madachy R, Medvidovic N, Meyer K, Meyers S, Perez G, Reinholtz KL, Roshandel R, Rouquette N, “Using Empirical Testbeds to Accelerate Technology Maturity and Transition: The SCRover Experience”, Proceedings of the 2004 International Symposium on Empirical Software Engineering, IEEE Computer Society, 2004

[Boehm-Lee 2005] Boehm B and Lee K, "Empirical Results from an Experiment on Value-Based Review (VBR) Processes," Proceedings, ISESE 2005, September 2005

[Boehm 2009] Boehm B, “Applying the Incremental Commitment Model to Brownfield System Development,” Proceedings, CSER 2009.

[Boehm-Lane] Boehm, B.W. and Lane, J.A, “Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering and Software Engineering”, Tech Report 2007-715, University of Southern California, 2007.

[Boehm-Lane 2010] Boehm B and Lane J, “"DoD Systems Engineering and Management Implications for Evolutionary Acquisition of Major Defense Systems," Proceedings, CSER 2010

[Bisignani-Reed 1988] Bisignani M., and Reed T, “Software Security Costing Issues”, COCOMO Users' Group Meeting. 1988. Los Angeles: USC Center for Software Engineering

[Booch 2009] Personal communication from Grady Booch, IBM, 2009

[Broy 2010] Broy M, “Seamless Method- and Model-based Software and Systems Engineering,” The Future of Software Engineering, Springer, 2010.

[Cohn 2005] Cohn M, Agile Estimating and Planning, Prentice Hall, 2005

[Colbert-Boehm 2008] Colbert E and Boehm B, "Cost Estimation for Secure Software & Systems," ISPA / SCEA 2008 Joint International Conference, June 2008

[DCARC 2005] Defense Cost and Resource Center, “The DoD Software Resource Data Report – An Update,” Proceedings of the Practical Software Measurement (PSM) Users’ Group Conference, 19 July 2005

[Galorath 2005] Galorath Inc., SEER-SEM User Manual, 2005

[Brooks 1995] Brooks, F., The Mythical Man-Month, Addison-Wesley, 1995.

[DSRC 2005] Defense Cost and Resource Center, “The DoD Software Resource Data Report – An Update,” Proceedings of the Practical Software Measurement (PSM) Users’ Group Conference, 19 July 2005.

[Elssamadisy-Schalliol 2002] Elssamadisy A. and Schalliol G., Recognizing and Responding to ‘Bad Smells’ in Extreme Programming, Proceedings, ICSE 2002, pp. 617-622

[Galorath-Evans 2006] Galorath D, Evans M, Software Sizing, Estimation, and Risk Management, Auerbach Publications, 2006

[Garmus-Heron 2000] Garmus, David and David Herron. Function Point Analysis: Measurement Practices for Successful Software Projects. Boston, Mass.: Addison Wesley, 2000

[Gill-Iansiti 1994] Gill G., and Iansiti M., “Microsoft Corporation: Office Business Unit,” Harvard Business School Case Study 691-033, 1994.

[Hopkins-Jenkins 2008] Hopkins R, and Jenkins K, Eating the IT Elephant: Moving from Greenfield Development to Brownfield, IBM Press.

[IFPUG 1994]. Function Point Counting Practices: Manual Release 4.0, International Function Point Users’ Group, Blendonview Office Park, 5008-28 Pine Creek Drive, Westerville, OH 43081-4899.

[IFPUG 2009] http://www.ifpug.org

[ISO 1999] ISO JTC 1/SC 27, Evaluation Criteria for IT Security, in Part 1: Introduction and general model, International Organization for Standardization (ISO), 1999.

[Jensen 1983] Jensen R, “An Improved Macrolevel Software Development Resource Estimation Model”, Proceedings of 5th ISPA Conference, 1983

[Koolmanojwong-Boehm 2010] Koolmanojwong S and Boehm B, "The Incremental Commitment Model Process Patterns for Rapid-Fielding Projects,"Proceedings, ICSP 2010, Paderborn, Germany.

[Kruchten 1998] Kruchten, P., The Rational Unified Process, Addison-Wesley, 1998.

[Lane 2009] Lane J., "Cost Model Extensions to Support Systems Engineering Cost Estimation for Complex Systems and Systems of Systems," 7th Annual Conference on Systems Engineering Research 2009 (CSER 2009)

[Lane-Boehm 2007] Lane J., and Boehm B., "Modern Tools to Support DoD Software Intensive System of Systems Cost Estimation - A DACS State-of-the-Art Report," August 2007.

[Lewis et al. 2008] Lewis G et al., “SMART: Analyzing the Reuse Potential of Legacy Components on a Service-Oriented Architecture Environment,” CMU/SEI-2008-TN-008.

[Li et al. 2009] Li Q, Li M, Yang Y, Wang Q, Tan T, Boehm B, Hu C, Bridge the Gap between Software Test Process and Business Value: A Case Study. Proceedings, ICSP 2009, pp. 212-223.

[Lum et al. 2001] Lum K, Powell J, Hihn J, “Validation of Spacecraft Software Cost Estimation Models for Flight and Ground Systems”, JPL Report, 2001

[Madachy 1997] Madachy R, Heuristic Risk Assessment Using Cost Factors”, IEEE Software, May 1997

[Madachy-Boehm 2006] Madachy R, Boehm B, “A Model of Options and Costs for Reliable Autonomy (MOCA) Final Report”, reported submitted to NASA for USRA contract #4481, 2006

[Madachy-Boehm 2008] Madachy R, Boehm B, “Comparative Analysis of COCOMO II, SEER-SEM and True-S Software Cost Models”, USC-CSSE-2008-816, 2008

[NCCA-AFCAA 2008] Naval Center for Cost Analysis and Air Force Cost Analysis Agency, Software Development Cost Estimation Handbook Volume 1 (Draft), Software Technology Support Center, September 2008

[Nguyen 2010] Nguyen V. "Improved Size and Effort Estimation Models for Software Maintenance," PhD Dissertation, Department of Computer Science, University of Southern California, December 2010, http://csse.usc.edu/csse/TECHRPTS/by_author.html#Nguyen

[Park 1988] Park R, “The Central Equations of the PRICE Software Cost Model”, COCOMO User’s Group Meeting, 1988

[Putnam and Myers 1992] Putnam, L.H., and W. Myers. Measures of Excellence. Prentice-Hall, Inc. Englewood Cliffs, NJ, 1992

[Putnam and Myers 2003] Putnam, L.H., and W. Myers. Five Core Metrics. Dorset House Publishing. New York, NY, 2003

[PRICE 2005] PRICE Systems, TRUE S User Manual, 2005

[QSM 2003] Quantitative Software Management, SLIM Estimate for Windows User's Guide, 2003

[Reifer et al. 1999] Reifer D, Boehm B, Chulani S, “The Rosetta Stone - Making COCOMO 81 Estimates Work with COCOMO II”, Crosstalk, 1999

[Reifer 2008] Reifer, D., “Twelve Myths of Maintenance,” Reifer Consultants, Inc., 2008.

[Reifer 2002] Reifer, D., “Let the Numbers Do the Talking,” CrossTalk, Vol. 15, No. 3, March 2002.

[Reifer 2002] Reifer D., Security: A Rating Concept for COCOMO II. 2002. Reifer Consultants, Inc.

[Royce 1998] Royce, W., Software Project Management: A Unified Framework, Addison-Wesley, 1998.

[Schwaber and Beedle 2002] Schwaber, K. and Beedle, M., Scrum: Agile Software Development, Prentice-Hall, 2002.

[Selby 1988] Selby R, “Empirically Analyzing Software Reuse in a Production Environment”, In Software Reuse: Emerging Technology, W. Tracz (Ed.), IEEE Computer Society Press, 1988

[Stutzke 2005] Stutzke, Richard D, Estimating Software-Intensive Systems, Upper Saddle River, N.J.: Addi-son Wesley, 2005

[USC 2000] University of Southern California Center for Software Engineering, “

[Tan et al. 2009] Tan,T, Li Q, Boehm B, Yang Y, He M, and Moazeni R, "Productivity Trends in Incremental and Iterative Software Development," Proceedings, ACM-IEEE ESEM 2009.

Guidelines for Model-Based (System) Architecting and Software Engineering (MBASE)”, available at http://sunset.usc.edu /classes_cs577b, 2000.

[USC 2006] University of Southern California Center for Software Engineering, Model Comparison Report, Report to NASA AMES, Draft Version, July 2006

[USD (AT&L) 2008] Systems Engineering Guide for System of Systems, Version 1.0, OUSD(AT&L), June 2008.

[Valerdi 2011] Valerdi R, Systems Engineering Cost Estimation with COSYSMO, Wiley, 2011.

[Yang et al. 2005] Yang Y, Bhuta J, Boehm B, and Port D, "Value-Based Processes for COTS-Based Applications," IEEE Software, Volume 22, Issue 4, July-August 2005, pp. 54-62